C++程序设计

（第2版）

和克智　编著

PROGRAMMING

西安交通大学出版社

C++程序设计

C++ CHENGXU SHEJI

（第2版）

和克智　　编著

西安交通大学出版社
·西安·

内 容 简 介

C++是 80 年代初出现的一种程序设计语言，它除了保持 C 语言的简洁高效等特点外，还支持面向对象的程序设计。

本书共分 15 章和 1 个附录，全面、系统地介绍了 C++语言的语言要素及 C++程序设计的方法与步骤。特别是在面向对象程序设计方面，书中通过逐步完成几个较大型的范例程序，从应用的角度予以详细的介绍，以期读者能够迅速地掌握这一新型的程序设计方法。

本书是 1995 年 7 月出版的《C++程序设计》一书的第 2 版，是作者集多年教学、科研和实际应用的心得体会而精心撰写而成的。书中许多内容是其它资料中没有提到而在实用中又会经常遇到的。相信读者能够从本书中得到较大的收获。

本书可用作大专院校 C++语言课程的教材，计算机软件开发和应用人员的参考资料，尤其适合自学 C++语言的读者使用。

（陕）新登字 007 号

图书在版编目（CIP）数据

C++程序设计 / 和克智编著. —2 版. —西安：西安交通大学出版社，1999.8

ISBN 7-5605-1141-4

Ⅰ.C… Ⅱ.和… Ⅲ.C 语言-程序设计 Ⅳ.TP312

中国版本图书馆 CIP 数据核字（1999）第 32143 号

*

西安交通大学出版社出版发行

（西安市咸宁西路 28 号 邮政编码 710049 电话：（029）2668316）

陕西省轻工印刷厂印装

各地新华书店经销

*

开本：787mm×1092mm 1/16 印张：21.875 字数：526 千字

1999 年 8 月第 2 版 1999 年 8 月第 1 次印刷

印数：1～5 000 定价：26.00 元

若发现本书有倒页、白页、少页及影响阅读的质量问题，请去当地销售部门调换或者与我社发行科联系调换。发行科电话：（029）2668357，2667874

前　言

目前风靡计算机世界的面向对象的程序设计方法是软件开发技术的一场革命。C++语言是一个优秀的面向对象的程序设计语言，它保持了C语言的简洁、高效和接近汇编语言的特点，消除了C语言中的一些不安全因素。更重要的是它将高科技研究中的面向对象的思维方式引进到程序设计语言中，增加了类（Class）这种语言成分以支持面向对象的程序设计。这不仅使得程序的可靠性和数据的安全性得到了保证，而且提高了代码的重用性，从而极大地提高了软件开发的效率，减轻了软件开发人员的劳动强度。

C++语言不仅支持面向对象的程序设计，同时也支持传统的面向过程的程序设计。实际上，由于C++语言的变量说明非常灵活，输入/输出操作极为方便，并提供了引用（Reference）数据类型，因此，用C++语言书写非面向对象的程序时也会比传统的程序设计语言方便得多。

本书是1995年7月出版的《C++程序设计》一书的第2版，全书共分15章和1个附录。书中以翔实的内容和精心编写的范例程序全面系统地介绍了C++语言的各个语言成分及其特性，以及利用C++语言进行程序设计的基本方法与设计要点。第1章介绍C++语言的发展及其特点、面向对象程序设计的基本概念、C++程序的基本结构、基本的输入/输出方法和开发C++程序的基本步骤。第2章介绍C++语言的词法符号、基本数据类型、变量、运算符和表达式。第3章介绍C++语言的程序控制语句。第4章介绍C++语言的函数、作用域和存储类以及编译预处理指令。其中，函数的重载在一个方面体现了面向对象程序设计语言的多态性。本章内容是全书的重点之一。第5章介绍C++语言的数组类型和字符串。第6章介绍C++语言的指针、引用和动态内存分配技术。本章内容也是全书的重点和难点之一。第7章介绍C++语言的结构、联合、枚举等数据类型以及类型定义。第8章介绍C++语言的类类型和对象。从本章起，书中的内容均与面向对象的程序设计密切相关。类体现了面向对象程序设计的封装性，因此本章内容也是全书的重点之一。第9章介绍C++语言中类的构造函数和析构函数。第10章介绍C++语言的继承和派生类，它们体现了面向对象程序设计的继承性，本章亦为全书的重点之一。第11章介绍C++语言中的友元、虚函数、静态成员以及const和volatile成员。其中，虚函数体现了面向对象程序的运行时的多态性。第12章介绍C++语言的运算符重载。运算符重载从另一个方面体现了面向对象程序的多态性，本章内容是全书的又一重点。第13章介绍C++语言的输入/输出流类。第14章介绍C++语言的文件流类和内存流类。这两章的内容不仅是全书的一个重点，同时也是一个难点。第15章介绍C++语言的模板。由于模板存在移植性较差的问题，因此本章可以作为选修内容。附录为ASCII码表。

书中每章后面都附有习题。应当说明的是：这些习题不仅可用来使读者巩固各章中的内容，它们同时又是相应章节内容的补充。由于篇幅所限，一些较大型的例子不便于在正文中给出，而是放在习题中由读者对它们进一步地完善。这一现象在第8章以后尤为明显。所以希望读者尽量完成所有的习题，以便全面地掌握C++语言。

本版为适应目前更新、更高的教学和学习要求，对第 1 版做了较大的改动：采用了最新的 ANSI C++ 2.1 标准；重新编写了书中的所有范例程序；增加对了模板的介绍；将文件流和内存流从原 13 章中分离出来自成一章，以便于更详细地介绍文件操作。由于不同的 C++语言具体实现总是存在一些差异，本版还对程序的移植性进行了说明。

本书是作者集多年对本科生、研究生的 C 语言和 C++语言的教学经验，并参阅大量中外资料的前提下撰写而成的。对学习 C++语言时常犯的错误和易混淆的概念进行了详细的介绍与提示。为使读者能更好地掌握 C++语言，作者根据多年在科研和教学中使用 C++语言的心得体会，精心编写了书中所有的范例程序并全部上机通过。另外还编写了几个较大型的程序，其内容贯穿 C++语言的主要语言成分，并在相应章节中逐步出现、逐步完善，以期为读者掌握面向对象的程序设计方法提供一个思路。应当指出的是：所有的范例程序均是利用 Borland C++ 5.02 编译器编译通过的，若读者使用的是其它编译器，可能会有个别的例子需要做少量的调整。

作者力求本书具有通俗性和实用性。尽量避免过多的理论描述，而用较多的篇幅介绍实用方法。书中的许多内容出自作者的经验和体会，是其它资料中所没有提到而在实用中又常常遇到的。作者将这些内容奉献给读者，期望读者通过本书能很快地掌握 C++语言，打消“C++语言难学”或“C++语言好学难用”的不正确看法。

本书将 C++语言作为一门独立的程序设计语言，而不是作为 C 语言的扩展来介绍的，因此，不要求读者先行掌握 C 语言。

本书可供大专院校作 C++语言教材使用，也可以作为从事计算机软件开发和应用人员的重要参考资料，尤其适合作为广大的计算机爱好者学习 C++语言的自学材料。将本书作为教材使用时，建议讲授学时不少于 60 学时。

对于本书存在的错误和不足之处，作者殷切希望广大读者批评指正。

和克智

1999 年 7 月

目　录

前言……(Ⅰ)

第1章　绪论……(1)
1.1　C++语言的发展和特点……(1)
1.2　面向对象的程序设计……(2)
1.2.1　抽象和封装——对象……(2)
1.2.2　派生和继承……(3)
1.2.3　多态性……(3)
1.3　C++程序的结构……(3)
1.4　基本的输入和输出……(5)
1.5　C++程序的开发步骤……(7)
习题……(8)

第2章　数据和表达式……(10)
2.1　C++语言的词法符号……(10)
2.1.1　关键字……(10)
2.1.2　标识符……(10)
2.1.3　标点符号……(11)
2.1.4　分隔符……(11)
2.2　基本数据类型……(11)
2.3　常量……(13)
2.3.1　数值常量……(13)
2.3.2　字符常量……(14)
2.3.3　字符串常量……(14)
2.4　变量及其说明……(15)
2.5　基本运算符和表达式……(16)
2.5.1　基本运算符……(16)
2.5.2　表达式……(21)
2.6　赋值表达式和类型转换……(22)
2.6.1　赋值表达式……(22)
2.6.2　增量减量运算符……(22)
2.6.3　复合赋值运算符……(24)
2.6.4　表达式中的类型转换……(24)
2.6.5　强制类型转换……(25)

2.7 简单变量的初始化……(26)
2.7.1 变量的初始化……(26)
2.7.2 const 和 volatile 修饰符……(27)
习题……(28)

第 3 章 程序控制语句……(29)
3.1 C++语言的语句……(29)
3.1.1 C++语句的基本形式……(29)
3.1.2 块语句……(29)
3.2 选择语句……(30)
3.2.1 if 语句……(30)
3.2.2 if 语句的嵌套……(32)
3.2.3 三元条件表达式……(33)
3.2.4 switch 语句……(34)
3.2.5 if 语句和 switch 语句的比较……(38)
3.3 循环语句……(38)
3.3.1 while 语句……(38)
3.3.2 do-while 语句……(40)
3.3.3 for 语句……(40)
3.3.4 三种循环语句的比较……(44)
3.3.5 循环的嵌套……(44)
3.4 循环的中断……(45)
3.4.1 break 语句……(46)
3.4.2 continue 语句……(46)
3.4.3 exit ( ) 函数和 abort ( ) 函数……(47)
3.4.4 补充说明……(48)
3.5 goto 语句与标号……(49)
习题……(50)

第 4 章 函数……(52)
4.1 函数……(52)
4.1.1 定义函数……(52)
4.1.2 函数的调用……(53)
4.2 函数间的数据传递……(53)
4.2.1 函数的参数和返回值……(54)
4.2.2 函数原型……(57)
4.2.3 函数的值调用……(58)
4.3 C++的库函数……(59)
4.4 作用域和存储类……(60)
4.4.1 作用域……(60)
4.4.2 存储类……(63)
4.5 函数的递归调用……(67)

4.6 内联函数……(70)
4.7 带有缺省参数的函数……(71)
4.8 参数数目可变的函数……(73)
4.9 函数重载……(75)
4.10 编译预处理……(77)
4.10.1 嵌入指令……(77)
4.10.2 宏……(78)
4.10.3 条件编译指令……(82)
4.11 程序的多文件组织……(84)
4.11.1 连接属性……(85)
4.11.2 分割编译……(87)
习题……(87)

第 5 章 数组……(89)
5.1 数组……(89)
5.1.1 一维数组……(89)
5.1.2 多维数组……(91)
5.1.3 数组间的赋值……(93)
5.1.4 数组与函数……(95)
5.2 字符串……(97)
5.2.1 字符串的存储形式……(97)
5.2.2 字符串数组……(98)
5.2.3 字符串间的赋值……(99)
5.2.4 字符串与函数……(99)
5.3 字符串处理库函数……(100)
习题……(102)

第 6 章 指针和引用……(103)
6.1 指针……(103)
6.1.1 指针概念……(103)
6.1.2 指针的说明……(103)
6.1.3 对指针的访问……(104)
6.1.4 指针的运算……(105)
6.1.5 多级指针……(108)
6.2 指针与数组……(108)
6.2.1 用指针访问数组元素……(109)
6.2.2 指针与字符串……(111)
6.2.3 指针数组和数组指针……(112)
6.3 指针与函数……(115)
6.3.1 指针作为函数的参数……(115)
6.3.2 返回指针的函数……(119)
6.3.3 C++语言中的 main ( ) 函数……(121)

6.3.4 指向函数的指针……(125)
6.4 指针和动态内存分配……(130)
6.5 引用……(133)
6.5.1 引用的说明与使用……(133)
6.5.2 引用与函数……(135)
6.6 void 和 const 型指针……(137)
6.6.1 void 型指针……(137)
6.6.2 const 型指针……(140)
习题……(141)

第 7 章 结构、联合和枚举……(143)
7.1 类型定义……(143)
7.2 结构……(144)
7.2.1 定义结构……(145)
7.2.2 说明结构变量……(145)
7.2.3 访问结构变量……(146)
7.2.4 结构变量成员……(148)
7.3 结构数组和结构指针……(150)
7.3.1 结构数组……(150)
7.3.2 结构指针……(152)
7.4 结构与函数……(153)
7.5 位域……(160)
7.6 联合……(162)
7.6.1 定义联合……(162)
7.6.2 联合的使用……(163)
7.7 枚举……(164)
7.7.1 枚举类型的定义及其变量的说明……(164)
7.7.2 枚举变量的使用……(166)
习题……(169)

第 8 章 类和对象……(171)
8.1 概述……(171)
8.2 类……(172)
8.2.1 类的说明……(172)
8.2.2 类与结构……(175)
8.2.3 内联成员函数……(175)
8.3 对象……(176)
8.3.1 对象的说明……(176)
8.3.2 对象的使用……(177)
8.3.3 类作用域……(179)
8.4 成员函数的重载……(181)
8.5 this 指针……(183)

习题……(184)

第 9 章　构造函数和析构函数……(185)
9.1　构造函数……(185)
9.1.1　定义构造函数……(185)
9.1.2　构造函数与对象初始化……(186)
9.1.3　构造函数和 new 运算符……(187)
9.1.4　缺省的构造函数……(188)
9.2　析构函数……(190)
9.2.1　定义析构函数……(190)
9.2.2　析构函数和 delete 运算符……(192)
9.2.3　缺省的析构函数……(193)
9.3　拷贝初始化构造函数……(195)
9.4　构造函数与对象成员……(200)
习题……(205)

第 10 章　继承和派生类……(206)
10.1　继承……(206)
10.1.1　单一继承……(207)
10.1.2　多重继承……(211)
10.2　初始化基类成员……(213)
10.3　二义性、支配规则和赋值兼容规则……(219)
10.3.1　二义性……(219)
10.3.2　支配规则……(221)
10.3.3　继承和对象成员……(222)
10.3.4　赋值兼容规则……(222)
10.4　虚基类……(223)
习题……(229)

第 11 章　类的其它特性……(230)
11.1　友元函数……(230)
11.1.1　友元函数的说明……(230)
11.1.2　使用友元函数……(231)
11.1.3　将成员函数用作友元……(231)
11.2　虚函数……(234)
11.2.1　虚函数……(234)
11.2.2　纯虚函数……(237)
11.3　静态成员……(243)
11.3.1　静态数据成员……(243)
11.3.2　静态成员函数……(245)
11.4　const，volatile 对象和 const，volatile 成员函数……(246)
11.5　指向类成员的指针……(251)

习题……(255)

第 12 章　运算符重载……(256)
12.1　运算符重载……(256)
12.1.1　重载运算符……(266)
12.1.2　使用运算符重载……(258)
12.1.3　友元运算符……(259)
12.1.4　转换函数……(262)
12.1.5　赋值运算符重载……(264)
12.2　几个特殊运算符的重载……(267)
12.2.1　增量减量运算符……(267)
12.2.2　下标运算符……(270)
12.2.3　其它几个特殊运算符的重载……(274)
12.3　字符串类……(275)
12.3.1　定义字符串类……(275)
12.3.2　使用字符串类……(279)
习题……(280)

第 13 章　C++语言的 I / O 流类……(282)
13.1　概述……(282)
13.1.1　流……(282)
13.1.2　文件……(283)
13.1.3　缓冲……(283)
13.2　C++的基本流类体系……(283)
13.2.1　基本流类体系……(283)
13.2.2　预定义的流及流运算符……(284)
13.2.3　流的格式化 I / O……(286)
13.2.4　流的错误处理……(293)
13.3　输入和输出……(296)
13.3.1　格式化输入……(296)
13.3.2　输入操作函数……(298)
13.3.3　格式化输出……(300)
13.3.4　输出操作函数……(301)
13.3.5　重载提取和插入运算符……(303)
习题……(304)

第 14 章　文件流和内存流……(305)
14.1　文件流……(305)
14.1.1　C++语言的文件流类体系……(305)
14.1.2　文件的打开与关闭……(306)
14.1.3　文件的访问……(308)
14.2　几个主要用于文件的函数……(310)

14.2.1 文件的读写函数……(310)
14.2.2 文件随机访问的函数……(313)
14.3 文本文件和二进制文件……(314)
14.3.1 文本文件……(314)
14.3.2 二进制文件……(318)
14.4 内存流……(321)
14.4.1 C++的内存流类体系……(321)
14.4.2 使用内存流……(322)
14.5 两个专门用于内存流的函数……(323)
习题……(325)

第 15 章 模板……(326)
15.1 函数模板……(326)
15.1.1 定义模板函数……(326)
15.1.2 使用模板函数……(328)
15.1.3 特例函数……(330)
15.2 类模板……(331)
15.2.1 定义模板类……(331)
15.2.2 使用模板类……(333)
习题……(335)

附录 ASCII 码表……(336)

参考文献……(337)

第1章 绪论

本章介绍 C++语言的起源、发展及其特点；面向对象程序设计的基本概念；C++程序的基本结构；C++的基本输入/输出功能。

1.1 C++语言的发展和特点

自 1946 年第一台电子计算机 ENIAC 问世以来，随着计算机应用领域的迅速扩大，硬件和软件都取得了高速的发展。作为计算机软件的基础，程序设计语言也得到不断的充实和完善。功能全面、使用便利的程序设计语言相继问世。

60 年代，Martin Richards 为计算机软件人员在开发软件系统时作为记述语言使用而开发了 BCPL 语言。1970 年，Ken Thompson 在继承 BCPL 的许多优点的基础上发明了比较实用的 B 语言。美国 DEC 公司的 PDP-7 型机中的 UNIX 操作系统就是用 B 语言记述和开发的。到了 1972 年，在美国贝尔研究所进行的对小型机 PDP-11 UNIX 操作系统的开发工作中，Dennis Ritchie 和 Brian Kernighan 对 B 语言作了进一步的充实和完善，推出了更加通用的 C 语言。C 语言具有以下的主要特点：

C 语言是结构化程序设计语言 结构化程序设计要求程序的逻辑结构只能由顺序、选择和循环三种基本结构组成。C 语言提供了编写结构化程序所需要的语句，十分有利于采用自顶向下、逐步求精的结构化程序设计技术。

C 语言是模块化程序设计语言 C 语言程序的函数结构非常便于把一个程序的整体分割成若干相对独立的功能模块，并且为程序模块间的相互调用以及数据传递提供了便利。

C 语言程序高效、灵活、功能强 C 语言是为了能够胜任系统程序设计的要求而开发的，具有很强的表达能力，能够用于描述系统软件各方面的特性。因此，用 C 语言编写的程序表述灵活，功能强大。另外，C 语言编写的程序生成的代码质量很高，所以 C 程序具有很高的运行效率。

C 语言可以部分地取代汇编语言 C 语言比较靠近硬件与系统，可以像汇编语言那样直接访问硬件。比如，C 语言的第一个应用就是开发 UNIX 操作系统，而该系统软件仅用了很少的汇编代码，绝大部分代码是用 C 语言写成的。

C 语言具有较高的移植性 在 C 语言中，没有依赖于硬件的输入 / 输出语句。在程序中，输入 / 输出操作是通过调用由系统提供的、独立于 C 语言的库函数来完成的。因此，C 语言本身并不依赖于计算机的硬件系统，从而便于在硬件结构不同的机器间实现程序的移植。

C 语言具有很强的数据处理能力 C 语言提供了丰富的运算符，除了可以对数据进行四则运算和逻辑运算外，还可以实现以二进制位（bit）为单位的诸多运算。另外，C 语言还提供了丰富的数据类型。因此，C 语言具有很强的数据处理能力。

由于 C 语言具有以上众多的特点以及 UNIX 的成功和广泛应用，使得 C 语言迅速得到普及，成为一种广为使用的程序设计语言。目前的各种机型和各种操作系统上都运行有 C 编译器。

然而，C 语言也存在着一些缺陷，比如：类型检查机制相对较弱；缺少支持代码重用的语言结构；不适合开发大型程序等。随着软件工程规模的扩大，C 语言的这些缺陷逐渐显露出来。比如，当程序源代码超过 50 000 行、开发人数达数十个时，系统维护工作量将变得相当大，而且系统的整体性能难以保证。

为了克服 C 语言存在的缺点，并保持 C 语言简洁、高效和接近汇编语言的特点，1980 年，贝尔研究所的 Bjarne Stroustrup 博士及其同事开始对 C 语言进行改进和扩充，把 Simula 67 语言中类的概念引入到 C 中，并将改进后的 C 语言称为“带类的 C”（C with class）。1983 年夏，由 Rick Maseitti 提议正式命名为“C++”，并于同年 7 月首次对外公开发表。在此之后，Stroustrup 博士与他的同事们在使用该语言所积累的经验的基础上，于 1985 年和 1989 年进行了两次修订，先后引进了运算符重载、引用、虚函数等许多特性，并使之更加精炼，于 1989 年底推出最新的 AT&T C++ 2.0 版。随后，美国标准化委员会以 AT&T C++ 2.0 为蓝本，制定了 ANSI C++标准。现在 ANSI C++已发展到版本 2.1，各种新版本 C++语言软件包都支持该标准，并有所扩展。

C++对 C 语言的扩充和改进是具有革命性的，这是由于 C++支持面向对象的程序设计。同时，C++又是 C 语言的一个超集，这就使得许多 C 代码不经修改就可以被 C++编译器编译通过。另外，用 C++编写的程序可读性更好，代码结构更为合理，可以更直接地在程序中映射问题空间的结构。更重要的是，C 程序员仅需学习 C++语言的新特征，很快就能够利用 C++语言来编写程序。

C++语言已经被应用于程序设计的众多应用领域，它尤其适合用于中型和大型的程序开发项目。有报告表明，C++已应用于 C 语言曾经使用过的所有场合，且其效果要比 C 语言好得多。从开发时间、费用到所形成的软件的可重用性、可扩充性、可维护性和可靠性等方面，都显示出 C++语言的优越性。

随着各种高效的软件开发平台的推出，C++语言也成为软件开发人员的必修和首选语言。比如 Microsoft 公司的 Visual C++，Borland 公司的 C++ Builder 等可视化开发平台，均要求开发人员具有熟练地使用 C++语言的能力。

1.2 面向对象的程序设计

面向对象的程序设计（Object-Oriented Programming，简称 OOP）是继承和发展了结构化程序设计而产生的一种新的程序设计思想，它是一种通过模仿人类建立现实世界模型的方法（包括概括、分类、抽象和归纳）进行软件开发的思想体系。面向对象的程序设计语言必须具有 4 种对象化属性：抽象性、封装性、继承性和多态性。

1.2.1 抽象和封装——对象

对象（Object）是 OOP 最重要的性质之一。从技术上讲，对象就是既含有数据（叫做对

象的属性——Attribute）又含有对数据进行操作的代码（叫做方法——Method）的一个逻辑实体。在对象中，有些属性和方法是为该对象所特有的，亦即不能为对象之外的任何东西直接访问的。这样，可以有效地防止程序中其它不相关部分无意地修改或不正确地使用这些属性和方法。

对象是类（Class）的实例（Instance）。类用于描述对象的群体特性，它同时又是进行抽象和封装的基石。

抽象（Abstraction）是指将具体事物一般化的过程。对具有特定属性及行为特征（即方法所执行的操作）的对象进行概括，从中提取这一类对象的共性，并从通用性的角度描述共有的属性及行为特征。抽象包括两方面的内容：一是数据抽象，即描述某类对象的公共属性；二是代码抽象，即描述某类对象的行为特征。通过类，可以很方便地实现数据抽象和代码抽象，这种结合方式就是所谓的封装（Encapsulation）机制。抽象和封装可以提高软件的模块化程度，增强代码的重用性。

访问一个方法的过程称为向一个对象发送一个消息（Message）。 对象的工作是靠消息来激发的，对象之间也是通过消息发生联系的，即请求其它对象做什么或响应其它对象的请求是通过发送和接收消息来实现的。

1.2.2 派生和继承

派生（Deriving）是指由基本类导出子类的过程，用来派生子类的类叫做基类，派生出的子类就叫做派生类。派生是分层次等级进行的。

继承（Inheritance）是指一对象获取另一对象之性质的过程。这与按层次分类的思想都是面向对象的主要性质。在有类这一概念之前，必须针对每个对象定义其所有性质；由于使用了类，就只需定义对象的区别于类中其它对象之性质。由于派生类的存在，那些与基类共享的性质就能很方便地被继承了下来。

派生和继承是代码重用的基础。

1.2.3 多态性

同一个消息为不同的对象所接收时，可能导致完全不同的行为，这样现象就称为多态性（Polymorphism）。多态性的重要性在于允许一个类体系的对象各自以不同的方式响应同一消息，即所谓的“同一接口，多种方法”。

多态性的优点在于：减轻了程序设计人员的记忆负担，代码的使用者只需要记住少数的几个接口就可以完成众多的操作；提高了程序设计的灵活性，程序中可以通过极为简单的操作来访问同一类体系中的不同对象。

1.3 C++程序的结构

学习任何一个程序设计语言，首先应了解用该语言所编写的程序之基本结构，这里，通过一个简单的程序来说明C++程序的结构及其特点。

例1.1 一个简单的C++程序。

```
//      EX1_1.CPP

/*      This program demonstrates the construction of a
        C++ program file.     */

#include      <iostream.h>

int     main()
{
        cout << "i = ";                 // Display a prompt
        int     i;                     // Declare variable i
        cin >> i;                      /* Get input for i */
        cout << "Your number is " << i << '\n';

        return 0;
}
```

当运行这个程序时，屏幕上将显示：

```
i =
```

并等待用户输入一个整数。设用户的输入为5，则屏幕上将显示：

```
Your number is 5
```

并返回操作系统。

程序中第1至4行是C++语言中的注释语句。与汇编语言相比，C++语言所编写的程序的可读性要好得多，然而，它和人类的自然语言仍有很大的距离。为了提高程序的可读性，在书写源程序时，应养成用注释来说明程序的功能、语句的作用的良好习惯。C++编译器在编译一个程序时，将跳过注释语句，不对它进行处理。因此，无论源程序中有多少注释，均不会增加可执行文件的长度。

C++语言提供了两种注释语句形式：一种是由符号“//”引出、“回车”结束的单行注释语句；另一种是由符号对“/*”和“*/”括起来的多行注释语句。前者主要用于较短小、内容不超过一行的注释，而后者主要用于较长的多行注释。当然，可以根据个人的爱好与习惯仅使用其中一种注释形式或将两种注释形式混合使用，如例1.1中第10至12行所示。

程序中第6行是一条编译预处理指令。关于编译预处理指令的详细内容将在第4章的4.10节中介绍。这里可暂时理解为：为了能进行输入/输出操作，程序中必须有这一行。

程序中以main开始的部分定义了一个函数，该函数规定了程序的功能。main是函数名，其后紧跟一对圆括号。所有的C++程序必须有且仅能有一个main函数，通常称该函数为主函数。一个C++程序总是从主函数中的第一条语句开始执行，在正常情况下，总是在执行完主函数中的最后一条语句后结束整个程序并返回操作系统。

一个C++函数中的所有内容均被括在一对花括号“{}”中，其中，左花括号表示“该函数从这里开始”，而右花括号表示“该函数到这里结束”。被括起来的内容叫做函数体(Function body)。函数体由一系列的C++语句所组成，这些语句描述了函数是怎样实现它的功能的。

在例 1.1 中，采用了比较整齐美观的所谓“缩进”格式来书写程序。事实上，C++程序的书写格式是非常自由的，比如，上例完全可以写成如下的形式：

```
//      EX1_1.CPP

/*      This program demonstrates the construction of a C++ program file. */
#include      <iostream.h>
int     main() {cout << "i = ";                  // Display a prompt
int     i;              // Declare variable i
cin >> i;       /* Get input for i */   cout <<"Your number is " << i <<     "\ n"; return 0;}
```

若主函数中没有单行注释的话，甚至可以将整个函数体全部连成一行。编译器是完全可以正确理解这个程序的。但对人来说，阅读这个程序实在是太困难了。

1.4 基本的输入和输出

一个程序常常会要求用户向它提供一些信息，程序接收外部信息的操作就叫做程序的输入。一个程序通常总是要向用户发出一些信息，程序向外部发送信息的操作就叫做程序的输出。C++程序输入操作可由 cin 对象来完成，而输出操作则可由 cout 对象来完成。

在 C++语言中，输入流 cin 和输出流 cout 具有极强的输入 / 输出功能和极为灵活的用法。这方面的内容将在第 13 章中作详细的介绍，这里，仅介绍它们最简单的用法。

cin 的基本用法具有如下的一般形式：

cin >> variable;

其中：variable 为一个变量（将在第 2 章中介绍）；“>>”称做提取运算符。这一行的意思是：程序暂时中止执行，等待用户从键盘上输入一个数据。若用户键入了一个有效的数据并按下回车键，程序就将用户键入的数据存入变量 variable 中，并继续执行程序的其它部分。比如，例 1.1 中的第 12 行：

```
cin >> i;
```

就是要求用户为变量 i 输入一个整数值。

C++语言允许用一个 cin 输入多个数据，比如：

```
cin >> i >> j;
```

就是要求用户分别为变量 i 和 j 各输入一个数据。

用户在为这种所谓串连式的 cin 输入数据时，应注意用空白字符（关于 C++的空白字符将在第 2 章中介绍）如空格将所输入的数据分隔开。例如，设用户欲为 i 输入 5，为 j 输入 10，则应键入：

5 10<CR>

其中，黑体字符为用户键入的数据，<CR>表示按下回车键（下同）。

应说明的是，当用户响应 cin 的要求输入数据时，必须注意所输入数据的类型（C++的数据类型将在第 2 章中介绍）应与接受该数据之变量的类型相匹配，否则输入操作将会失败

或者得到的将是一个错误的数据。

cout 的基本用法具有如下的一般形式：

cout << expression;

其中：expression 是一个表达式（第 2 章中介绍）；“<<”称做插入运算符。这一行的意思是：将表达式 expression 的值输出（显示）到屏幕上当前光标位置处。比如，设 i 的值为 5，则

```
cout << i;
```

将在屏幕上当前位置处显示 5。

cout 还可以输出常数和用双引号引起来的一串文字，例如：

```
cout << "Here will display a ten ";
cout << 10;
```

将在屏幕上显示：

```
Here will display a ten 10
```

与 cin 相同，C++允许用一个 cout 输出多个数据，比如：

```
cout << i << j;
```

将使得变量 i 和 j 的值依次显示在屏幕上。

但应注意的是，使用串连式输出时，cout 将欲输出的多个数据项连续地显示出来，各数据项之间不加任何分隔符。例如，设 i 的值为 5，j 的值为 10，则上述输出为

```
510
```

因此，用 cout 进行多项输出时，程序的设计者（以下简称程序员）有责任人为地加入一些分隔符，以免输出产生二义性。在进行屏幕输出时，常用的分隔符有空格和水平制表符，它们可分别用 ' ' 和 '\t' 表示。前者表示在两个输出项之间插入一个空格；后者表示下一个输出项应显示在屏幕上的下一个显示区（Zone）。屏幕上的一个显示行可以分为多个显示区，通常，一个显示区的宽度为 8 个字符宽。这就是说，第一显示区从屏幕上的第 1 列开始；第二显示区从屏幕上的第 9 列开始；第三显示区从屏幕上的第 17 列开始；其余依此类推。在输出时，一旦遇到水平制表符，则下一个输出项将从当前光标所在显示区的下一个显示区开始显示。但若当前光标正好位于一个显示区的第一列时，即上一个输项出恰好占满了前一个显示区，则下一个输出项将再推后一个显示区。

通常，屏幕的宽度为 80 个字符宽。当输出的内容超过第 80 列时，显示会自动地转到下一行的行首。但是，这种系统自动控制的显示方式常常使得输出显得比较混乱，不利于用户阅读输出的内容。C++语言为 cout 提供了新行指示符，使得程序员可以控制输出在新的一行显示，使输出美观易读。

C++提供了两种新行指示符：'\n' 和 endl。两者的作用完全相同，都是使随后的输出内容从新的一行开始显示。同一功能用两个符号表示似乎有点多余，事实上，前者是 C 语言中的新行符，C++只所以保留它是为了保持与 C 语言的兼容；而后者则是 C++语言的 I / O 流类操纵符（关于流类及其操纵符将在第 13 章中介绍）。

下面用一个例子来演示上述的内容。

例 1.2 cout 演示程序。

```
//      EX1_2.CPP

#include      <iostream.h>

int     main()
{
        int     i, j;
        long    k;

        i = 5;
        j = 10;
        k = 12345678;

        cout << "12345678123456781234567812345678\n";
        cout << i << ' ' << j << endl;
        cout << i << '\t' << j   << '\n';
        cout << k << '\t' << j << endl;
}
```

该程序的输出为：

```
12345678123456781234567812345678
5 10
5       10
12345678        10
```

输出中的第 1 行是为了指示出各个显示区的位置，第 1 行输出的两个数据用空格分隔，第 3，4 行输出的两个数据用水平制表符分隔。在第 4 个输出行中，由于前一个数据正好占满一个显示区，所以后一个数据被显示在第 3 显示区中，

1.5 C++程序的开发步骤

C++语言是一种编译性语言。一个 C++源程序需经过编译、连接，生成可执行文件，然后在操作系统的支持下直接键入可执行文件名就可以运行该程序。一个 C++程序的开发过程大致可分为以下 4 个步骤：

第 1 步　编辑源程序　源程序就是程序员按照 C++语法规定而书写的程序代码。例 1.1 和例 1.2 中的程序代码均是源程序。

程序员首先应将源程序输入计算机，并为它起一个名字以便存盘时使用。文件的名字由两部分组成：第一部分叫做文件名，它由不超过 8 个英文字母或数字或下划线构成；第二部分叫做文件扩展名，它由不超过 3 个英文字母或数字构成。这两部分用英文的句号“.”隔开。习惯上，C++的源程序的文件扩展名取为 CPP，而 C++头文件（第 4 章中介绍）的扩展名取为 H。例如：

EX1_1.CPP

MYPROG.CPP
EX1_1.H

输入源程序可以借助各种字处理程序，然而，现在大多数 C++语言软件包都带有一个集成环境，比如 Borland C++软件包的 IDE，Microsoft C++的 WB 等，这些集成环境的基本界面就是一个文字处理器。使用这些软件包的集成环境，程序员不需要借助其它的字处理程序就可以编辑源程序了。

第 2 步　编译　将编辑好了的源程序利用 C++软件包所提供的编译器进行编译，生成扩展名为 OBJ 的目标文件。不同的 C++软件包配有不同的编译器，其使用方法也各不相同。若使用软件包中的集成环境，则只需在主菜单中选择“编译”菜单项即可。

在编译过程中，编译器可能会发现源程序中的一些错误，这时它就会给出相应的错误信息。在这种情况下，程序员应再回到编辑器中，改正源程序中的错误。反复这一过程，直至编译通过。

第 3 步　连接　用连接器将编译成功的目标文件与相应的启动文件、库文件等连接成扩展名为 EXE 的可执行文件。有的操作系统本身就提供有连接器，比如 MS-DOS 的 Link，但最好还是使用 C++软件包所提供的连接器。不同的 C++软件包所提供的连接器的使用方法也各不相同。若使用软件包中的集成环境，则只需在主菜单中选择“连接”菜单项即可。

许多 C++软件包的集成环境中都有一个 Make 功能，选择该功能，可以使得编译与连接一次完成。

连接过程中还可能发现一些源程序在编译时发现不了的错误，这时，仍应退到编辑器中，改正错误。再编译、连接，直至连接成功，磁盘上就会生成一个可以直接执行的.EXE 文件。

第 4 步　调试　编译连接成功的程序不一定就是一个正确的程序，它只说明了源程序的语法是正确的、应连接的模块都是存在的，决不能保证程序运行的结果一定正确。例如，程序员的本意是要求两个数的和，却误将加号敲成了减号；或者所采用的算法本身就是错误的，等等。这些错误叫做程序的逻辑错误。编译器和连接器一般是检查不出逻辑错误的。所以，对一个编译连接成功的程序一定要用适当的数据对其进行调试，尽可能地发现其中的逻辑错误并改正之。

习题

1-1　选择一个 C++软件包，熟悉它的使用方法。

1-2　将本章中的例子输入到计算机中，编译并运行这些程序。

1-3　下面是一个含语法错误的源程序。编译该程序，看一看编译器会给出些什么错误信息。根据错误信息进行修改，直至成功地生成可执行文件。并执行该程序。

```
main
{
    int   i;          /* This is a variable
    int   j;          // This is another variable

    i = 5
    j = 10;
```

```
cout << "i + j = << i + j;
```

1-4 C++一般不允许在一条注释中再嵌套另一条注释，例如：

```
/* This is a /* inlay */ comment */
```

常常会被编译器当做一个错误。但许多 C++的具体实现（编译器）中则允许嵌套注释。试将这一行加入到已修改正确的习题 1-3 中，看一看你所使用的编译器是否允许嵌套注释。

第 2 章　数据和表达式

本章介绍 C++语言中的词法符号、基本数据类型、常量、变量、运算符和表达式。

2.1　C++语言的词法符号

词法符号是程序中最小的、不可再分的语法单位。C++语言中共规定了 6 种词法符号：关键字、标识符、常量、字符串、运算符和标点符号。本节介绍关键字、标识符和标点符号，其它词法符号将在以后的章节中分别介绍。

2.1.1　关键字

关键字是 C++语言的保留字，它们在 C++语言中具有特定的含意，在程序中不得将它们作为它用。ANSI C++ 2.1 规定了近百个关键字，加之各种 C++软件包均有不同程度的扩充，一个具体的 C++软件包（通常称其为一个具体实现）所指定的关键字常常多达一百二三十个。下面仅列出本书中将要介绍的 44 个关键字：

auto　break　case　char　class　const　continue　default　delete　do　double
else　enum　extern　flaot　for　friend　goto　if　inline　int　long　new
operator　private　protected　public　register　return　short　signed　sizeof
static　struct　switch　template　this　typedef　union　unsigned　virtual
void　volatile　while

这 44 个关键字为各种版本 C++软件包所支持，使用它们所编写的 C++程序具有很好的移植性。

2.1.2　标识符

标识符是程序员为程序中的一些要素所起的名字，比如变量名、函数名等。C++语言规定标识符可以由字母、数字和下划线组成，并规定第一个字符必须是字母或下划线。比如：

```
mystr      GetValue_123      double_list      binary_tree
```

均为合法的标识符。而以下的标识符均属于非法的：

```
2_tree      // 起始字符不附合规定
chr$        // 含有非法字符$
friend      // 关键字
```

C++语言属于大小写敏感语言，即认为字母的大写与小写是有区别的。比如，下面的标识符将被编译器看作合法的、且是不同的标识符：

Friend　　FRIEND　　fRIENE　　frienD

各种 C++语言标准对标识符的长度没有明确的规定，但在大多数具体实现中一般均规定标识符的长度不超过 32 个字符。即，若两个标识符的前 32 个字符的拼法完全相同，则编译器将认为它们是同一个要素的名字。

合理地使用大小写与下划线来拼写标识符，可以有效地提高程序的可读性。

2.1.3 标点符号

C++语言中的标点符号有 9 个：

#　　(　　)　　{　　}　　,　　:　　;　　...

在程序中的某些地方必须使用某种标点符号，但不表示任何操作。不同标点符号的具体用法将在后续章节中逐步介绍。

2.1.4 分隔符

在书写源程序时，一个词法符号与另一个词法符号必须用某种符号将它们分隔开来。用于分隔词法符号的符号就叫作分隔符。

在 C++语言中，标点符号和运算符均可以用作分隔符。比如：

```
int    i;      float  f;

j = i + 5;
```

然而，最常使用的分隔符是空白字符。在 C++语言中，空白字符指的是空格键、回车键和 Tab 键所产生的字符。其中 Tab 键与水平制表符 '\t' 的作用相同。源程序的缩进书写格式通常就是利用 Tab 键来实现的。

另外，C++编译器将注释语句也当作空白对待，所以，注释也可以用作分隔符。

2.2 基本数据类型

在 C++程序中，所有数据均具有某种数据类型（type）。类型限定了数据的存储方式和可以对它们实施的操作。C++语言中，数据类型又分基本数据类型和导出数据类型。基本数据类型是 C++语言预定义的类型，而后续章节将要介绍的数组、指针、结构、联合、枚举和类等导出数据类型，则是程序员根据实用的需要按照 C++语法的规定由基本数据类型构造出来的。C++的基本数据类型如表 2.1 所示。

注意：表 2.1 中所给出的数据的长度和取值范围均是指在 16 位操作系统下的值，而在 32 位操作系统中，数据长度和取值范围基本上都比表中给出的数值大一倍（下同）。

字符型变量用于存储 ASCII 码字符，也可存储 8 位的二进制数；整型变量用于存储整数；

实型和双精度型变量用于存储实数。无值型将在第 6 章中介绍。

表 2.1 C++语言的基本数据类型

数据类型	名称	长度（bit）	取值范围
char	字符型	8	-128～127
int	整型	16	-32 768～32 767
float	实型	32	3.4E-38～3.4E+38
double	双精度型	64	1.7E-308～1.7E+308
void	无值型	0	无值

除无值类型外，基本数据型还可以带有各种修饰前缀用于改变基本数据类型的含义，以适应不同情况下的需要。C++类型修饰符共有 4 种，如表 2.2 所示。

表 2.2 类型修饰符

修饰符	意义
signed	无符号
unsigned	有符号
long	长
short	短

其中，signed 和 unsigned 可以用来修饰字符型和整型数据，long 和 short 可以修饰整型数据，long 还可以修饰双精度型数据。表 2.3 给出了 C++中所有合法的基本数据类型和类型修饰符的组合。

表 2.3 C++中所有基本数据类型和类型修饰符的组合

类型	长度（bit）	取值范围
char	8	取决于具体实现
signed char	8	-128～127
unsigned char	8	0～255
short int	16	-32 768～32 767
signed short int	16	-32 768～32 767
unsigned short int	16	0～65 535
int	16	-32 768～32 767
signed int	16	-32 768～32 767
unsigned int	16	0～65 535
long int	32	-2 147 483 648～2 147 483 647
signed long int	32	-2 147 483 648～2 147 483 647
unsigned long int	32	0～4 294 967 295
float	32	3.4E-38～3.4E+38
double	64	1.7E-308～1.7E+308
long double	80	3.4E-4932～1.1E+4932

当用这 4 种修饰符来修饰 int 时，关键字 int 可以省略，比如：

short——→short int
signed——→signed int
unsigned——→unsigned int
long——→long int
unsigned long——→unsigned long int

另外，无修饰前缀的 int 被编译器缺省认为是有符号的，即 signed int。而对 char，则不

同的具体实现有着不同的规定：有的编译器缺省为无符号的，有的编译器缺省为有符号的。实型数据永远是有符号的。

2.3 常量

常量是用来表示固定不变的数值或字符的词法符号。在程序中，常量不经任何说明就可以直接使用。常量也具有一定的数据类型，但在程序中并不需要用数据类型关键字来专门说明，其表示方式自动地决定了它们的数据类型。

2.3.1 数值常量

C++语言中的数值常量有整型数值常量和实型数值常量两种。

2.3.1.1 整型常量

整型常量可以表示成十进制、八进制和十六进制的形式。十进制整型常量由 0～9 这 10 个数字字符组成，第一个字符不得是 0；八进制整型常量由 0～7 这 8 个数字字符组成，第一个字符必须是 0；十六进制整型常量由 0～9 及 A～F 这 16 个字符组成，且必须以 0X 打头，其中所有的字母字符大小写均可且意义不变。整型常量的前边可以加正负号表示它们值的正负，正号可以省略不写。

下面是几个整型常量的例子：

72304　　0440　　0x8342　　-578　　+087357　　-0X4bf　　2678345097

整型常量的数据类型将根据它的值按下列顺序来匹配：

十进制	int, long, unsigned long
八进制	int, unsigned, long, unsigned long
十六进制	int, unsigned, long, unsigned long

按此规则，上述例子中的 7 个常数的数据类型应分别为：

long　　int　　unsigned　　int　　unsigned long　　int　　unsigned long

可以为一个整型常量后缀一个 L（或 l），使其数据类型满足 long 或 unsigned long；可以为一个整型常量后缀一个 U（或 u），使其数据类型满足 unsigned 或 unsigned long；也可以同时后缀 U 和 L，使其数据类型满足 unsigned long。例如：

38975L　　0234l　　5678u　　0X91U　　543lu　　0247UL

2.3.1.2 实型常量

实型常量又称实常量。实型常量只能使用十进制形式表示，又可分为一般表示形式和指数表示形式。一般表示形式和我们日常数学中实数的表示方法相同；指数表示形式由尾数部分、字母 e 或 E 和指数部分组成：

234.78e-27
尾数　指数

其中，指数部分必须是一个整数。若指数大于零，可以省略其前边的正号。

2.3.2 字符常量

字符常量是一个单一的字符，它的基本表示方式是将一个字符用一对单引号括起来。例如：

'a'　　'A'　　'8'　　'?'　　'$'　　' '（空格）

均是 C++语言中的字符常量。其中，单引号只是用来说明其间的字符为字符常量，即起一个定界符的作用，它本身不是字符常量的内容。在 C++语言中，这种用单引号括起来的单一字符所表示的字符常量的数据类型为 char 型。

在 C++语言中，还有一种表示字符常量的方式，即所谓的“转义序列”。转义序列就是用转义字符“\”后跟一个字符或一个整型常量来表示一个单一的字符。若转义字符的后边是一个整型常量，则该常量必须是一个无起首 0 的八进制或十六进制数，例如：'\101'，'\x41' 和 'A' 均表示字符常量 A。由于 ASCII 码的值为 0～255，所以，转义字符后边的整型常量若为八进制数，则不得超过 3 位；若为十六进制数，则不得超过 2 位。

在实用中，转义序列通常被用来表示那些不可显示的或是从键盘上无法输入的字符。表 2.4 列出了 C++语言预定义的转义序列字符和它们的含义。

表 2.4 转义序列字符及其含义

表示	功能
\a	响铃
\b	退格（Backspace 键）
\f	换页
\n	新行
\r	回车
\t	水平制表（Tab 键）
\v	垂直制表
\\	反斜杠
\'	单引号'
\"	双引号"

表 2.4 中最后 3 个字符虽然既可显示又可从键盘上输入，但由于它们在 C++语言中具有特殊的意义：“\”是转义字符；“'”用于表示字符常量；而“"”则用于表示字符串常量，所以，当它们作为字符常量时，必须采用转义序列。

应当指出的是，在 C++语言中，转义序列所表示的字符常量的数据类型为 int 型。

2.3.3 字符串常量

字符串常量是用一对双引号括起来的字符序列，例如：

"This is a character string."

与单引号在字符常量中的作用相同，双引号在这里也仅是作为定界符，它本身不是字符串常量的内容。若在字符串常量中需要出现双引号则必须使用转义序列。例如：

"I say: \"Here need a\".\""

所表示的字符串为

I say: "Here need a"."

在 C++语言中，字符串的存储方式比较特殊。一个字符串常量并不是一系列字符常量的

简单排列，系统会在每一个字符串常量的尾部加上一个字符常量 '\0'，表示“该字符串常量到此结束”。字符常量 '\0' 是一个值为 0 的 ASCII 字符，在 ASCII 码表中，称它为 NUL。

例如，字符串常量 "C++ language" 共有 12 个字符，但它要占据 13 字节的存储单元，如图 2-1 所示（其中的数字是各个字符 ASCII 码的十六进制值）。因此，一个字符串常量所占的存储单元总是比它的字符个数多一个字节。

43	2b	2b	20	6c	61	6e	67	75	61	67	65	0

图 2-1　字符串常量的存储形式

应当指出的是，在字符串常量中使用转义序列时，要注意防止出现二义性。例如：

```
cout << "\x7Error!";
```

其原意是先使计算机的喇叭响一声，然后显示“Error!”6 个字符。但实际执行的结果是在屏幕上显示：

~rror!

且无喇叭响声。

这是由于编译器将 7 和 E 连在一起，将它们解释成 '\x7E'，而十六进制的 7E 正好是字符 '~' 的 ASCII 码值，于是就会出现以上的现象。这时，就是把\x7 改写成\x07 也不能改变这种现象，这是由于 C++将转义序列看作是一个整型数据，转义序列甚至可以由 4 位十六进制数组成。

此例可以采用以下方法来解决：

```
cout << "\x7 Error!";
```

即用一个空格将转义序列字符与其它字符分隔开来，明确地将它们分为两个词法单元。或者干脆将它们分成两个常量输出：

```
cout << '\x7' << "Error!";
```

2.4　变量及其说明

一个变量就是内存中一块用一个标识符命名的存储单元，用于存放可由程序修改的数据。命名一个变量的标识符就称做该变量的变量名。由于 C++是一种编译性语言，所以任何一个变量必须在使用前进行说明，以便编译器为它预留存储单元。

变量说明的一般形式为：

<storage> type var_name<, var_name...>;

其中：storage 是变量的存储类，关于存储类将在第 4 章中介绍；type 为任一合法的 C++数据类型；var_name 是一个标识符，即程序员为变量所起的变量名。说明中用尖括号括起来的部分是可选项，即在实际应用中这一项可有可无（在本书以后的介绍中，除了在预处理指令中使用的尖括号外，若无特别说明，均为此义）。例如：

```
int    i;                  // 说明了一个名为 i 的整型变量
char   c, str;             // 说明了两个变量名分别为 c 和 str 的字符型变量
long double ld;            // 说明了一个名为 ld 的长双精度实型变量
```

虽然 C++语言像其它编译性语言一样，要求一个变量必须“先说明后使用”，但 C++的变量说明具有更大的灵活性，即只要求变量在使用它之前甚至在使用它的同时说明即可，并不像有些语言——比如 C 语言——那样，变量说明必须集中地放在一个程序模块的起始部分。例 1.1 中变量 i 的说明就演示了 C++语言变量说明的这种灵活性。

2.5 基本运算符和表达式

变量、常量通过运算符组合成 C++语言的表达式，从而构成 C++程序的基本要素。本节介绍 C++语言中的基本运算符和表达式。

2.5.1 基本运算符

运算是对数据的加工过程，记述各种不同运算的符号叫做运算符，而参与运算的数据叫做操作数。不同的语言有不同的运算符集，表 2.5 给出了 C++语言的运算符集。

表 2.4 C++语言的运算符

优先级	运算符	结合性
1	() -> [] :: .* ->* &（引用）	右结合
2	*（递引用） &（取地址） new delete ! ~ ++ -- -（负号） sizeof	左结合
3	* / %	右结合
4	+ -	右结合
5	<< >>	右结合
6	< <= > >=	右结合
7	== !=	右结合
8	&（按位与）	右结合
9	^	右结合
10	|	右结合
11	&&	右结合
12	||	右结合
13	?:	左结合
14	= += -= *= /= %= <<= >>= &= ^= |=	左结合
15	,	右结合

运算符的优先级与我们日常数学中意义相同，它决定了一个表达式的运算顺序。运算符的结合性则决定了该运算符对其操作数的运算顺序：如果一个运算符对其操作数至左向右地执行规定的运算，则称该运算符是右结合的；反之则称其为左结合的。

按运算符所要求操作数的个数，运算符可以分为一元运算符、二元运算符和三元运算符。一元运算符要求有一个操作数，二元运算符要求有两个操作数，而三元运算符则要求有 3 个

操作数。按运算符的运算性质，运算符又可以分为算术运算符、关系运算符、逻辑运算符、位运算符和其它运算符。本节介绍 C++语言中的算术运算符、关系运算符、逻辑运算符、位运算符和 sizeof 运算符，其余的运算符将在后继章节中逐步介绍。

2.5.1.1 算术运算符

C++语言中的算术运算符包括一元算术运算符

－（负号）

和二元算术运算符：

＋（加）　　－（减）　　＊（乘）　　／（除）　　％（模除）

一元算术运算符又叫做一元减运算符，其作用与我们日常数学中的负号的作用相同，即取操作数的负值。二元算术运算符“+”，“－”与我们日常数学中的加、减号的作用相同，而“*”，“/”则分别对应日常数学中的乘号和除号。二元算术运算符“%”叫做模除运算符，其作用是取被除数对除数的模，即被除数除以除数后的余数。比如，13%5 的结果为 3。

运算符“%”要求它的两个操作数必须都是整型或字符型数据，而其它算术运算符则可以是任何基本数据类型。但应当指出的是，若运算符“/”的两个操作数均为整型数据时，则运算结果即商也是一个整型值，小数部分被自然舍弃了，比如，13/5 的结果为 2，而 5/13 的结果为 0。

与算术运算符有关的一个问题是计算中的溢出处理问题。在 C++语言中，被零除和实数溢出将被视为一个严重错误而导致程序运行的异常终止。但整数溢出却不认为是一个错误，其原因在于有时需要利用整数溢出进行一些较低层的程序设计，比如，查看设备的状态位、设计一个伪随机数发生器程序，等等。因此，程序员有责任检查和处理整型数据算术运算中的溢出问题。比如，设有下面的程序片断：

```
//...
int    i, j;

i = 32767;
j = i + 1;
cout << "i = " << i << '\t' << "j = " << j << endl;
//...
```

期望的输出是

```
i = 32767    j = 32768
```

然而实际的输出却是

```
i = 32767    j = -32768
```

我们知道，变量在内存中是以二进制形式存储的，变量 i 的值 32767 的二进制形式为

0111 1111 1111 1111

当该值加 1 以后，将变成如下的形式：

1000 0000 0000 0000

若作为无符号数，该值就是十进制的 32768；若作为有符号数，该值则是十进制数 - 32768 的补码。由于 j 是一个 int 型的变量，它是有符号的，所以就会出现上述的结果。

整型数据进行减法运算时，亦会出现类似的情况。而在进行乘法运算时，溢出的可能性就更大。因此，在设计一个程序时，应预先估计一下运算结果的数量级，以便正确地设置变量的数据类型，防止溢出的产生。例如，可将变量说明成 long 型，甚至可以说明成 long double 型。

2.5.1.2 关系运算符

C++语言中的关系运算符都是二元运算符，它们一共有 6 个：

< （小于） <= （不小于） > （大于） >= （不大于） == （等于） != （不等于）

关系运算符是用来对两个操作数进行关系比较的。由于信息在计算机中均是以二进制数的形式存储的，因此，关系比较实质上就是比较两个操作数的大小。由于 C++语言中没有提供逻辑型数据，所以关系运算的结果是一个整型数：当两个操作数满足关系运算符所要求的比较关系时，其结果为 1，相当于逻辑真；否则为 0，相当于逻辑假。正因为如此，关系运算的结果可以用在算术运算中，例如：

```
i = (3 < 5) + 8;
```

则 i 的值为 9。

由于算术运算符的优先级高于关系运算符，所以，上例中若去掉圆括号，编译器将把它解释为

```
i = 3 < (5 + 8);
```

这时，i 的值将为 1。

关系运算符的两个操作数可以是任何基本数据类型，但是在比较实数的相等时，往往会出现判断错误，这是由于实数在计算机中的存储方式和计算误差所引起的。例如，有两个实型变量 g 和 f，经过一系列的运算后，设它们的值在理论上都应当是 5.0。然而由于计算的误差，它们的实际值可能分别是 4.999999 和 5.000001，从而使得关系运算 g == f 的结果是 0。因此，在判断两个实数是否相等时，应当判断它们之差的绝对值是否小于一个给定的极小的数，例如，对于上例，以下运算：

```
fabs(g - f) < 1e-5
```

的值将为 1。其中，fabs()是 C++语言中求实数绝对值的系统函数。

注意 6 个关系运算符中除大于和小于外的 4 个运算符的写法，它们都是双字符符号，键入这样的符号时，两个字符间不得有任何分隔符。另外注意等于关系运算符，它由两个等号组成，与赋值运算符（将在 2.6 节中介绍）形式不同。而在有些语言中——如 BASIC 语言和 Fortran 语言，等于运算符与赋值运算符则具有完成相同的形式。

2.5.1.3 逻辑运算符

C++语言中的逻辑运算符包括一元逻辑运算符

!（逻辑非）

和二元逻辑运算符：

&&（逻辑与）　　||（逻辑或）

逻辑运算符是用来表示两个操作数的逻辑关系的。与关系运算的结果相同，逻辑运算的结果也是用整型数据 1 和 0 来表示逻辑真和逻辑假的。逻辑运算的结果同样可以作为一个整型数值用于算术运算中。

1. **逻辑非**　当操作数的值为 0 时，对该操作数逻辑非运算的结果为真；否则为假。例如：设整型变量 i 和 j 的值分别为 0 和 - 30，则 !i 和 !j 的运算结果分别为 1 和 0。

2. **逻辑与**　只要两个操作数中有一个操作数的值为 0，则对它们逻辑与运算的结果就为假。例如，设 i 和 j 仍为上述值，则 i && j 的运算结果为 0。

3. **逻辑或**　只要两个操作数中有一个操作数的值不为 0，则对它们逻辑或运算的结果就为真。例如，设 i 和 j 仍为上述值，则 i || j 的运算结果为 1。

逻辑运算符的操作数可以是任何基本数据类型的数据。

2.5.1.4　位运算符

C++语言中的位运算符包括一元位运算符

~（位求反）

和二元位运算符：

&（位与）　　|（位或）　　^（位异或）　　<<（左移）　　>>（右移）

位运算符是对其操作数按其二进制形式逐位地进行逻辑运算或移位操作的。

为了下面举例的方便，现设有两个变量：

```
unsigned char     c, d;
```

并设它们的值分别为 135 和 43，则它们的二进制表示分别为 1000 0111 和 0010 1011。

1. **按位求反**　运算符“~”将其操作数逐位地取其反码，即将原来为 1 的位变为 0，原来为 0 的位变为 1。例如，按位求反~c 的结果为 0111 1000，即十进制 120。

2. **按位与**　运算符“&”将其两个操作数对应位逐一地进行逻辑与运算。两个一位二进制数逻辑与运算的规则为“见零则零”，即两个数中只要有一个为 0，则其逻辑与的结果就为 0。例如，按位与运算 c & d 的结果为 0000 0011，即十进制的 3。这一运算可以用竖式表示为：

```
c          1000 0111
d          0010 1011
c & d      0000 0011
```

3. **按位或**　运算符“|”将其两个操数作对应位逐一地进行逻辑或运算。两个一位二进制数逻辑或运算的规则为“见壹则壹”，即两个数中只要有一个为 1，则其逻辑或的结果就为 1。例如，按位或运算 c | d 的结果为 1010 1111，即十进制的 175。这一运算可以用竖式表

示为：

```
c          1000 0111
d          0010 1011
c | d      1010 1111
```

4. **按位异或** 运算符“^”将其两个操数作对应位逐一地进行逻辑异或运算。两个一位二进制数逻辑异或运算的规则为“同则零，异则壹”，即两个数只要不同，则其逻辑异或的结果就为 1；否则为 0。例如，位运算 c ^ d 的结果为 1010 1100，即十进制的 172。这一运算可以用竖式表示为：

```
c          1000 0111
d          0010 1011
c ^ d      1010 1100
```

5. **按位左移** 运算符“<<”将其左操作数向左移动其右操作数所指定的位数，移出的位补以 0。由于移位运算符的右操作数表示的是欲移动的位数，所以，它必须是一个整型表达式。例如，位移运算 d << 1 的结果为 0101 0110，即十进制的 86；而位移运算 d << 2 的结果为 1010 1100，即十进制的 172。从这两个例子可以看出：将一个数左移一位，相当将该数乘以 2；左移两位，相当将该数乘以 4。一般说来，将一个数左移 n 位，就相当将该数乘以 2^n。所以，在程序中，常用左移位来进行快速的乘法运算。

用移位方法进行乘法运算时，同样要注意溢出问题。例如：c << 1 的结果为 14，而不是 c 的两倍 270，这是由于移位时将 c 的最高位移出之故。同时还应注意，若被移位的是一个有符号数，移位后可能使该数的符号发生变化，这一点对右移运算同样适用。

6. **按位右移** 运算符“>>”将其左操作数向右移动其右操作数所要求的位数，移出的位补以 0。例如，位移运算 c >> 2 的结果为 0010 0001，即十进制的 33。与左移操作相对应：将一个数右移 n 位，相当于将该数除以 2^n，其小数部分将被忽略。这与整型和字符型数据的除法运算完全一致，所以在程序中常用右移位来进行快速的除法运算。

根据位运算符的操作性质可见，位运算符的操作数的数据类型必须是整型或字符型，不得是实型数据。

注意：按位左移运算符与插入运算符同形，按位右移运算符与提取运算符同形。那么编译器是如何来区分 cout << 3 是将 cout 左移 3 位还是向屏幕上输出一个 3 呢？编译器将把该表达式解释成向屏幕输出 3。这是由于 cout 是 C++语言中预定义的输出流类的对象，在该对象所属的类中，对运算符“<<”所执行的操作进行了新的定义（运算符重载。将在第 12 章中介绍）。对于按位右移与提取运算符，编译器也是按此原则解释。

2.5.1.5 sizeof 运算符

sizeof 运算符是一个一元运算符，它是用来计算其操作数在内存中所占的字节数的，其运算结果是一个无符号整型值。sizeof 运算符的用法具有以下的一般形式：

sizeof(type)

其中，type 是任一合法的 C++数据类型，包括前面介绍过的基本数据类型和以后将要介绍的导出数据类型。C++语言提供 sizeof 运算符是为了提高程序的移植性。因为在不同的机器上或 C++语言的不同实现中，以及不同的操作系统中，同一数据类型所占的内存大小不尽相同。

利用该运算符，可以使得程序员在涉及到内存计算时，不必考虑机器的具体型号和语言的具体实现以及所使用的操作系统。例如，在 16 位操作系统下——如 MS-DOS，Windows 3.X，sizeof(char)，sizeof(int)和 sizeof(double)的运算结果分别为 1，2 和 8；而在 32 位操作系统下——如 Unix，Windows NT，Windows 95，Windows 98，上述的运算结果则分别为 1，4 和 16。

2.5.2 表达式

将常量、变量用运算符连接起来的式子叫做表达式。作为表达式的特例，一个单独的变量或常量也可以是一个表达式。下面是几个表达式的例子：

```
a + b – c * 3 + d / e
a > b && c < d || e >> 3 != 0
a
35
```

2.5.2.1 表达式的运算顺序

C++语言中的所有操作都是用表达式表示的，所以，清楚地了解表达式的求值顺序是正确书写表达式的关键。

表达式的求值顺序取决于表达式中参与运算之运算符的优先级、结合性和语言的具体实现。例如，表达式：

```
a + b * c
```

的运算顺序为：先计算 b * c 的值，然后再将它们的乘积与 a 的值加起来。再如：

```
a * –b
```

由于一元运算符“–”的结合性为左结合的，所以该表达式的运算顺序为：先对 b 其负值，然后再与 a 相乘。

由于 C++语言的运算符数量庞大，优先级、结合性各不相同，记忆起来有一定的难度。所以，适当地在表达式中加上一些圆括号，不仅明确地规定了表达式的运算顺序，而且提高了程序的可读性。例如表达式：

```
(a > b) && (c < d) || ((e >> 3) != 0)
```

实际上并没有改变原表达式的运算顺序，但却方便了人们对该表达式的理解。C++编译器在编译源程序时，会自动删除源程序中的分隔符，所以多加的圆括号对并不会影响程序的运行效率。

2.5.2.2 逗号表达式

逗号“,”既是一个 C++的标点符号，又是一个 C++的运算符。作为运算符，它被叫做逗号运算符。用逗号运算符连接起来的表达式就叫做逗号表达式。例如表达式：

```
a + b, b * c, c – a
```

就是一个逗号表达式。

逗号表达式自左向右地计算各个分表达式的值，而整个逗号表达式的值为其最右端那个

分表达式的值。若设 a = 3，b = 5，c = 7，则上述逗号表达式的值为 4。

上例没有什么实用意义，实际上，在 C++程序中，逗号运算符常用来将多个赋值表达式（下节介绍）连成一个逗号表达式，例如：

```
a = a + b, b = c * 4, c = c - a
```

设 a，b，c 的原值仍为上述所设，则经过该表达式的运算后，a 的值为 8，b 的值为 28，c 的值为-1，且整个表达式的值也为-1。更进一步：

```
x = (a = a + b, b = c * 4, c = c - a)
```

则 x 的值为-1。

2.6 赋值表达式和类型转换

在程序中使用最为频繁的表达式就是赋值表达式，在 C++语言中，赋值的形式又非常灵活多变。本节介绍 C++语言中的赋值表达式和表达式中的类型转换。

2.6.1 赋值表达式

用赋值运算符“=”连接起来的表达式叫做赋值表达式。赋值表达式是 C++语言（包括 C 语言）中所特有的，其它大多数语言中，只有赋值语句而没有赋值表达式这一语法要素。

赋值表达式具有以下的一般形式：

variable = expression

其中：variable 是一个变量；而 expression 是一个合法的 C++表达式。

赋值表达式的作用就是将赋值运算符右操作数的值——expression 的运算结果——赋给左操作数 variable。该值同时又是整个赋值表达式的值。

由于赋值运算符的右操作数可以是任一合法的 C++表达式，而赋值表达式是一个合法的 C++表达式，所以，赋值表达式可以作为赋值表达式的右操作数。例如：

```
i = j = 5
```

就是一个合法的赋值表达式，它相当于：

```
i = (j = 5)
```

即先把 5 赋给变量 j，然后把赋值表达式 j = 5 的值——也是 5——赋给变量 i。

2.6.2 增量减量运算符

C++语言中提供了两个含有赋值运算意义的算术运算符，它们就是增量运算符和减量运算符：

++（增量）　　--（减量）

增量运算符和减量运算符都是一元运算符，其运算结果是将其操作数的值加一或减一。

例如，设 i 的值为 5，则++ i 后，i 的值将变为 6。即表达式++ i 相当于赋值表达式 i = i + 1。

由于增量减量运算符本身就隐含有赋值操作，所以它们的操作数必须是一个变量，且只能是整型或字符型变量。

然而，与其它一元运算符不同的是，这两个一元运算符的操作数既可以放在运算符的前边，又可以放在运算符的后边：

```
++ i（前置增量）
i ++（后置增量）
-- i（前置减量）
i --（后置减量）
```

作为一个单独的表达式时，运算符前置和后置是没有什么区别的。但在一个还包含有其它运算的表达式中，运算符前置和后置却具有不同的含义。例如，设 i 的值为 5，则进行赋值运算

```
j = ++ i
```

后，i 的值为 6；j 的值也为 6。而在进行赋值运算

```
j = i ++
```

后，则 i 的值为 6，j 的值却为 5。

由上例可见：在一个还包含有其它运算的表达式中，增量或减量运算符的前置意味着先对其操作数进行相应的增量或减量运算，然后再进行其它的运算；而增量或减量运算符的后置则意味着先进行其它的运算，然后再对其操作数进行相应的增量或减量运算。

在上节中曾经介绍过，表达式的运算顺序还会受过语言的具体实现的影响。这一点主要表现在增量和减量运算符上。例如，设变量 i 的值为 5，则输出语句：

```
cout << i ++ << '\t' << i ++ << endl;
```

的输出结果可能是

```
5    6
```

也可能是

```
6    5
```

这完全取决于具体的编绎器是先对第一个增量表达式求值还是先对第二个增量表达式求值。

解决这个问题最好的方法是不要将多个对同一变量的增量或减量表达式放在同一个表达式或语句中，比如，将上述输出语句改写为两个输出语句：

```
cout << i ++ << '\t';
cout << i ++ << endl;
```

则无论用哪种编绎器进行编绎，输出结果都是相同的：

```
5    6
```

2.6.3 复合赋值运算符

C++语言中的所有二元算术运算符和位运算符均可以与赋值运算符组合成一个单一的运算符——复合赋值运算符，它们一共有 10 个：

+=（加等）	-=（减等）	*=（乘等）	/=（除等）	%=（模除等）
&=（与等）	|=（或等）	^=（异或等）	<<=（左移等）	>>=（右移等）

复合赋值运算符可以用来构成复合赋值表达式，其构成形式与赋值表达式的一般形式相同。复合赋值表达式的运算过程为：先将两个操作数做运算符所规定的算术或位运算操作，然后将运算的结果赋给其左操作数。下面以运算符“*=”为例说明复合赋值表达式的运算过程。

设 i 和 j 的值分别为 5 和 6，则：

```
i *= i + j
```

表示先计算复合赋值运算符的右操作数——算术运算表达式——的值，即 i 与 j 的和，其运算结果为 11；然后，做复合赋值运算符所规定的算术运算，即求 i 与上述结果的乘积，其运算结果为 55；最后进行赋值运算操作，即将上述运算结果赋给复合赋值运算符的左操作数 i。同时，整个表达式的值亦为 55。其它复合赋值运算符所构成复合赋值表达式的运算过程以此类推。

2.6.4 表达式中的类型转换

C++语言允许在一个表达式中存在有不同数据类型的操作数。对这样的表达式求值时，编译器会对其中的一些操作数自动地进行类型转换（称为隐式类型转换），以使一个二元运算符两个操作数的类型一致。

隐式类型转换的规则为：

1. 所有的 char 和 short 都被转换成 int（unsigned 除外），所有的 float 都被转换为 double。

2. 对于所有的操作数对，若其中一个操作数为 long double，则另一个也被转换成 long double；否则，若其中一个操作数为 double，则另一个也被转换成 double；否则，若其中一个操作数为 long，则另一个也被转换成 long；否则，若其中一个操作数为 unsigned，则另一个也被转换成 unsigned。

例如，设 i 为一整型变量，则表达式：

```
i + 1.5
```

中二元算术运算符“+”的两个操作数的类型不一致——一个为整型，一个为实型。所以，按照上述规则，编译器将会自动地将它们转换成 double 型后再求和，其运算结果，即和的数据类型亦为 double 型。

应当说明的是：类型的转换仅是临时性的，它并不改变变量的数据类型，只是在执行运算的时刻，将变量的值做了转换。比如，对上例来讲，进行过上述运算后，变量 i 仍保持它的整型数据类型。正因为如此，在含有赋值运算的表达式中，有可能发生精度的丧失。例如：

```
int    i, j;

i = 3;
j = i * 0.3;
cout << j << endl;
```

则其输出结果是 0。

出现这样的结果是因为：尽管按照隐式转换规则，赋值运算符的右操作数（即表达式 i * 0.3）的值为一 double 型的 0.9，但由于这一转换是临时性，进行过赋值运算后，变量 j 仍保持其整型数据类型，从而导致其小数部分自然被截去。

2.6.5 强制类型转换

虽然隐式转换有它一定的优越性，但是，由于程序员无法控制表达式的类型，进而可能影响表达式最终结果的精度。例如：

```
int    i;
float  f;

i = 5;
f = i / 4 + 20.3;
cout << f << endl;
```

则其输出结果为 21.3，而不是预计的 21.55。这是因为编译器在处理赋值运算符右边的表达式时，按照运算符的优先级，先进行 i / 4 的计算。由于算术运算符“/”两个操作数的数据类型一致，没有必要进行类型转换，从而使得计算的结果为一整型数，其值为 1。因此，导致最终的计算结果少了一个 0.25。

为了解决类似的问题以及提高程序设计的灵活性，C++语言提供了所谓的强制类型转换（cast）手段，其一般形式为：

(type)expression 或 ***type(expression)***

例如，欲将一个整型变量 i 强制转换成实型的，则可采用以下方式：

(float)i 或 float(i)

在习惯上，人们更喜欢使用后者。这是因为这一表达方式很像 C++语言的函数调用，同时这一表达方式对于复杂表达式的强制转换显得更加自然。例如，欲将表达式 i * j + k 的运算结果强制转换成 double 型，若使用前者，则必须写成

```
(double)(i * j   + k)
```

若少了后边的一对括号，编译器就会将其解释成仅将变量 i 强制转换成 double 型。

有了强制转换这一手段，我们就可以使上例得到一个满意的结果：

```
int    i;
```

```
float  f;

i = 5;
f = double(i) / 4 + 20.3;                    // A
cout << f << endl;
```

由于 i 被强制转换成了 double 型，根据隐式转换规则，整数 4 也被编译器转换成 double 型，除法运算的结果也是 double 型。最终将使 f 获得正确的结果——21.55。

强制类型转换虽然清晰易读，但显得有些繁琐，在一些特殊的场合下还可以采用更为简便的方法。就上例而言，将语句 A 改为

```
f = i / 4.0 + 20.3;
```

同样可以获得正确的结果，这实际上是利用了隐式转换规则。

2.7 简单变量的初始化

一个变量必须有一个具体的值才有实用意义。这可以用赋值操作为其置上一个值，也可以在说明变量的同时为其赋予一个初值，后一方式称为变量的初始化。

2.7.1 变量的初始化

简单变量初始化的一般形式为

<storage> type variable = constant_expression;

其中，constant_expression 是一个与被说明之变量的类型 type 相匹配的、具有确定值的表达式。例如：

```
int    i = 5;
char   ch = 'a' + i;
```

由于变量 i 被初始化为 5，从而使得表达式'a' + i 有了确定的值——102（字符 a 的 ASCII 码为 97）——也就是字符 f 的 ASCII 码。

C++语言中，变量的初始化还可以延伸到导出类型的变量以及类的对象，还可以用一个有返回值的函数调用来初始化变量。关于这些内容，将在以后相应的章节中逐一介绍。

上述的初始化变量的一般形式与 C 语言的中初始化变量的形式完全相同，实际上以 C++语言中还有一种变量初始化的形式：

<storage> type variable(expression);

例如，上例可以改写为：

```
int    i(5);
char   ch('a' + i);
```

但在实用中，这种形式通常被用来初始化一个类的对象，很少用在简单变量上。

2.7.2 const 和 volatile 修饰符

C++语言中的关键字 const 和 volatile 是一种类型修饰符，它们可以用在变量的说明中。

当一个变量被关键字 const 所修饰时，这个变量的值就再也不允许被修改了，在 C++语言中，称该被修饰的变量名为常量标识符。正因为如此，一个被 const 所修饰的变量必须进行初始化。例如：

```
const int     i = 5;
```

就将 i 说明为一个常量标识符。在程序中，i 可以像其它变量一样地使用，但它却不得用作赋值表达式的左操作数，因为对一个变量赋值就意味着要重置它的值。

常量标识符在程序中主要用于作函数参数的场合，以阻止函数内部对它的修改（这一点将在第 5 章中介绍），因此，它具有与常量不同的作用。

被关键字 volatile 所修饰的变量称为易变标识符，它的作用是提醒编译器该变量的值可以通过程序中未明确定义的方法来改变。例如，当程序中的一个变量被映射到设备所使用的内存空间（例如显示缓冲区），或为独立的进程（例如中断例程）所共享时，这些设备或独立的进程可能在任何时刻修改这个变量的值。如果 C++编译器不知道一个变量的值会受到它所在程序以外之独立事件的影响，将可能对这个变量的使用采取一定的优化策略：它总是假定在一段程序中，如果一个变量不出现在赋值运算符的左边，那么这个变量的值在这段程序中就一直保持不变。因此，编译器将优化对该变量的使用方法，以尽量提高对它的访问效率。例如，下面的程序片断用于计算一段程序执行时所需的时间：

```
//.....
unsigned long      clock;

time1 = clock;
//.....
time2 = clock;
deff = time2 - time1;
```

变量 clock 用于记录系统时钟的当前时间。clock 由一个时钟中断程序来修改，系统时钟每经过一秒，clock 的值就被加 1，而上述程序仅使用该变量的值。对这个程序片断，由于 clock 从未出现在赋值运算符的左边，因此 C++编译器将对它的使用作这样的优化：从内存中读取 clock 的值，并将该值存入一个内部寄存器中供以后使用这个值时直接从该寄存器中读取。经过这样的优化，事实上 time1 和 time2 取得的是同一个值。因为若在上述两条语句之间发生过时钟中断，时钟中断程序就会修改 clock 的值，但内部寄存器中的值却没有修改，而 time2 取得的正是这个未修改的值，这将导致变量 deff 的值为 0。因此，为了取得正确的值，变量 clock 必须使用 volatile 进行修饰：

```
volatile unsigned long    clock;
```

这样，编译器就不会对 clock 作上述的优化。程序中每当要读取 clock 的值时，是在内存中而不是在内部寄存器中读了。

这两个关键字可以同时用来修饰一个变量，例如：

```
const volatile int   i;
```

说明变量 i 是 const 的，在程序中不得将它用于赋值运算符的左边；同时，它又是 volatile 的，即它的值可能由硬件或操作系统内部来改变。

在以后的章节中我们还可以了解到，这两个关键字还可以用于函数、对象，等等。

习题

2-1 试指出下列常量的数据类型。

```
65400       234u       123.45       1.5e52       3L       45Ul
'\73'       'a'        "A string"   "a"          "\33\s2f"
```

2-2 编写一个程序，测试一下你所使用的 C++编译器缺省地将 char 型数据设置成有符号的还是无符号的。

2-2 试写出下列表达式的值。

(1) 5 > 8 || 3 < 5　　(2) !(4 < 10)

(3) 'a' > 33　　(4) 'e' – 'a' == 5

(5) !10 < 5　　(6) !0 && 5

2-3 设 x = 3，y = 4，z = 5，则下列表达式的值各为多少？

(1) x | y & z　　(2) x & y | -z

(3) x ^ y ^ z　　(4) x | y && ~z

2-4 设 x = 22，则下列表达式的值各为多少？

(1) x ^ x　　(2) x >> 5

(3) x << 3　　(4) x += 33

(5) x >>= ~x　　(6) x %= 5

2-5 编写一个程序，根据用户输入的 3 条边长值计算三角形的周长和面积。

2-6 编写一个程序，由用户输入两个整数 val1 和 val2，输出稍大于 val1 且是 val2 整倍数的数。其计算公式为

```
val1 + val2 – val1 % val2
```

2-7 编写一个程序将用户输入的华氏温度转换成摄氏温度。其计算公式为

$$C = (F - 32) \times 5 / 9$$

第3章　程序控制语句

程序控制语句控制了整个程序执行的流程。从某种意义上讲，程序控制语句是任何一种算法语言的精髓。不同的程序执行控制方式确定了一种语言的风格。C++语言中提供了灵活丰富的程序控制语句，这也是C++语言倍受欢迎的原因之一。本章将详细地介绍C++语言中的各种程序控制语句。

3.1　C++语言的语句

语句是程序中可以独立执行的最小单元。语句通常是由表达式构成的，但表达式不是语句，所以表达式不能在程序中独立存在。

3.1.1　C++语句的基本形式

在C++语言中，语句的基本形式就是一个表达式后跟一个分号“;”来构成的。例如：

```
a = b + c;
i + j;
;
```

均是C++语言的合法语句。

在上述语句中，第1条语句是一条由赋值表达式构成的语句，通常称其为赋值语句；第2条语句是一条由算术运算表达式构成的语句，应当说明的是，该语句尽管是合法的，但没有什么实际意义，一般在程序中不使用这样的语句；第3条语句是由一个空的表达式构成的语句，这样的语句叫做空语句。空语句不仅合法，而且在程序中有其实用价值，它常被用于在程序中某处根据语法要求应该有一条语句，而实际上又没有什么操作可执行的场合。

在C++语言中，变量的说明必须以分号结束。所以变量的说明也是语句，叫做说明语句。正是由于在C++语言中存在说明语句这一概念，才使变量说明可以出现在程序中任何一个语句可以出现的地方，提高了变量说明的灵活性。对比其它语言，比如C语言，由于说明不是语句，所以变量只能集中地在一个模块的首部进行说明。

3.1.2　块语句

由一对花括号“{}”括起来的多条语句叫做一个块语句。例如：

```
{
    int   i = 10;
```

```
    i *= i;
    cout << i << endl;
}
```

块语句也叫复合语句，它在语法上等价于一条语句。块语句主要用于这样的场合：在程序的某处需要执行一项必须由多条语句才能完成的操作，而从语法上讲，程序在此处只允许存在一条语句。花括号是 C++语言中的一个标点符号，左括号标明了块语句的起始位置，右括号标明了块语句结束，它的作用就像单条语句中的分号一样。因此，右花括号后边不再需要分号。

3.2 选择语句

选择语句又叫分支语句，它控制程序流程的方式是：根据给定的条件，决定执行多个分支程序段中的哪一支。

3.2.1 if 语句

if 语句又叫做条件分支语句，它的功能是根据给定的条件，决定执行两个分支的哪一个分支。

3.2.1.1 基本的 if 语句

基本的 if 语句具有如下的一般形式：

```
if(expression)
    statement;
```

其中：expression 是一个关系或逻辑表式；而 statement 是一条合法的 C++语句，叫做 if 语句的语句段。语句段可以是一个单条语句，或是一个块语句，甚至是一个空语句。它的执行过程是：如果表达式的值为真，则执行语句段；否则就跳过语句段，直接执行 if 语句的后继语句。例如：

```
if(a > 10)
    a -= 5;

a += 5;
```

该例的执行过程是这样的：如果 a 的值大于 10，则从 a 中减去 5，然后执行语句 a += 5；否则直接执行语句 a += 5。这里，第 1，2 行一起构成了一条 if 语句。再如：

```
if(a > b)    {
    int    t = a;

    a = b;
    b = t;
}
```

```
cout << "The maximum is: " << a << endl;
```

该例的执行过程为：若 a 大于 b，则执行块语句中的 3 条语句，（它们实际上是交换 a 和 b 的值），然后执行输出语句；若 a 不大于 b，则直接执行输出语句。

3.2.1.2 扩展的 if 语句

扩展的 if 语句的一般形式为：

```
if(expression)
        statement1;
else
        statement2;
```

它的执行过程是：若表达式成立，则执行语句段 $statement_1$，否则执行语句段 $statement_2$。通常，将前者叫做 if 分支，将后者叫做 else 分支。

应当说明的是：尽管扩展的 if 诗句中存在两个语句段，且有两个表示语句结束的分号，但整个语句在语法上只是一条语句。尤其要注意 if 分支后边的分号是不可缺少的（除非这里是一条复合语句）。而在有的语言——比如 Pascal 语言——中，$statement_1$ 后边则不得有分号。

例 3.1 从键盘上输入两个整数，输出其中大者。

```
//      EX3_1.CPP

#include      <iostream.h>

int     main()
{
        int     a, b;

        cout << "Please enter two integers: ";
        cin >> a >> b;
        cout << "The maximum is ";

        if(a > b)
                cout << a << endl;
        else
                cout << b << endl;

        return 0;
}
```

该程序要求用户输入两个整数，然后输出较大那个数的值。在这里，if 语句起到决定输出哪个值的作用。

再看一个例子：

例 3.2 二次方程求根程序。

```
//      EX3_2.CPP

#include      <iostream.h>
#include      <math.h>                    // A
```

```
int     main()
{
        double          a, b, c, delta;

        cout << "Enter factors a, b and c: ";
        cin >> a >> b >> c;

        delta = b * b - 4 * a * c;
        if(delta < 0)
                cout << "No solution !\n";
        else    {
                delta = sqrt(delta);
                double          x1 = (-b + delta) / (2 * a);
                double          x2 = (-b - delta) / (2 * a);

                cout << "X1 = " << x1 << '\t' << "X2 = " << x2 << endl;
        }

        return 0;
}
```

本例中，在 else 分支中使用了复合语句。

程序中的 sqrt（）是一个 C++语言的库函数，它要求用一个 double 型的数据作其参数，并返回一个 double 的值。程序中的行 A 是为了能使用该函数所必需的。关于 C++库函数及其使用方法将在第 4 章中介绍。

3.2.2 if 语句的嵌套

由于 if 语句的语句段要求是一条 C++语句，而 if 语本身就是一条合法的 C++语句，所以可以将 if 语句用作 if 语句的语句段。这就是所谓的 if 语句的嵌套。

例 3.3 输出 3 个数中的最大数。

```
//      EX3_3.CPP

#include      <iostream.h>

int     main()
{
        cout << "Enter three integers: ";

        int     a, b, c;

        cin >> a >> b >> c;

        cout << "The maximum is: ";

        if(a >= b)
                if(a >= c)
                        cout << a << endl;
```

```
        else
                cout << c << endl;
    else
        if(b >= c)
                cout << b << endl;
        else
                cout << c << endl;

    return 0;
}
```

从此例可以看出，无论是 if 分支还是 else 分支，其语句段都可以是 if 语句。

例 3.3 中有诸多的 if 和 else，那么，哪个 else 与哪个 if 配对构成一条扩展的 if 语句呢？原则是“就近”，即一个 else 与离它最近的那个 if 配对。在实际应用中，为了提高程序的可读性，常常人为地将复杂的 if 嵌套写成复合语句的形式。比如，将例 3.3 中的 if 嵌套改写成如下的形式，程序的结构就显的更加清晰：

```
if(a >= b)  {
        if(a >= c)
                cout << a << endl;
        else
                cout << c << endl;
}
else  {
        if(b >= c)
                cout << b << endl;
        else
                cout << c << endl;
}
```

3.2.3 三元条件表达式

C++语言中提供了一个三元运算符“?:”，它的用法具有以下的一般形式：

expression ? expression$_1$: expression$_2$

其中：expression 是一个关系或逻辑表达式；expression$_1$ 和 expression$_2$ 均是任何合法的 C++ 表达式。

三元运算符用来构造 C++语言（包括 C 语言）中所特有的三元条件表达式。三元条件表达式的运算过程为：若 expression 的值为真，则整个表达式的值为 expression$_1$ 的值；否则为 expression$_2$ 的值。例如表达式：

```
max = a > b ? a : b
```

其含义为：若 a 大于 b，则将 a 的值赋给 max；否则将 b 的值赋给 max。它与以下的 if 语句等效：

```
if(a > b)
```

```
        max = a;
    else
        max = b;
```

三元表达式与 if 语句的区别在于：前者是一个表达式，只有构成语句后才可以在程序中独立出现；而后者则是一条语句。

在程序设计中，常用三元表达式构成的语句来代替简单的 if 语句，以简化程序的设计及提高运行效率。

例 3.4 例 3.3 的另一版本。

```
//      EX3_4.CPP

#include      <iostream.h>

int     main()
{
        cout << "Enter three integers: ";

        int     a, b, c;

        cin >> a >> b >> c;
        cout << "The maximum is: ";
        cout << (a >= b ? (a >= c ? a : c) : (b >= c ? b : c)) << endl;

        return 0;
}
```

该例中最后一条输出语句与例 3.3 中嵌套的 if 语句的作用完全相同，即，若 a >= b 为真，则输出表达式（a >= c ? a : c）的值；否则输出（b >= c ? b : c）的值。而表达式（a >= c ? a : c）的值为 a 和 c 中大者的值，表达式（b >= c ? b : c）的值为 b 和 c 中大者的值。

3.2.4 switch 语句

switch 语句又叫开关选择语句，它的功能是根据给定条件的状态，决定执行多个分支的哪一个分支。

开关选择语句具有以下的一般形式：

```
switch(expression)      {
        case  value1:
              statement1;
              <break;>
        case  value2:
              statement2;
              <break;>
              .
              .
              .
```

```
        case  value_n:
              statement_n;
              <break;>
        <default:
              statement_n+1;
              <break;>>
    }
```

其中：expression 是一个值为整型或字符型的 C++表达式，而各个 $value_i$ 则是不相同的整型或字符型常量值。

switch 语句的执行过程为：若某个 case 后边常量 value 的值与表达式 expression 的值相等，则执行该 case 后边的所有语句，直到遇见 break 语句为止。若表达式的值与任何一个 value 的值都不相等，那么：若存在 default 分支，则执行该分支后边的所有语句，直到遇见 break 语句为止；否则什么都不执行。

由 switch 语句的执行过程可以看出，若某一分支要求执行的是不止一条语句，则所要求执行的多条语句不必写成块语句的形式。

注意：default 分支可以放在 switch 语句中的任何位置，以满足不同的需要。

例 3.5 一个简单的计算器程序。

```
//      EX3_5.CPP

#include      <iostream.h>

int     main()
{
        int     num1, num2;
        char    oper;

        cout << "Enter a sample arithmetic expression: ";
        cin >> num1 >> oper >> num2;
        switch(oper)  {
              case    '+':
                      cout << (num1 + num2) << endl;

                      break;

              case    '-':
                      cout << (num1 - num2) << endl;

                      break;

              case    '*':
                      cout << (num1 * num2) << endl;

                      break;

              case    '/':
```

```
            cout << (num1 / num2) << endl;

            break;

        default:
            cout << "Operator you entered is illegal!" << endl;

            break;
    }

    return 0;
}
```

该**程序**提示用户输入一个简单的算术表达式，然后根据输入的表达式中运算符的不同，对所输入的两个数进行相应的运算，并输出运算结果。本**程序**只能处理两个数的加、减、乘、除运算，若所输入的运算符不是上述 4 种运算符之一，则输出一条错误信息：

Operator you entered is illegal!

设用户对**程序**的响应为

Enter a sample arithmetic expression: **9 - 3**<CR>

则**程序**的输出为

6

程序中各分支的 break **语句**是必要的，它指示**程序**在执行完相应分支中的**语句**后，就退出 switch **语句**。若将例中的 break **语句**全部删除，而用户对**程序**的响应仍为上述的输入，则**程序**的输出将变为：

6
27
3
Operator you entered is illegal!

在对 EX3_5 **程序**进行输入时，所输入的 3 个数据必须用空白字符隔开，如**第 1 章** 1.4 节所述。

例 3.6 根据用户所输入的月份，输出从 1 月 1 日到给定月末的总天数。

```
//      EX3_6.CPP

#include      <iostream.h>

int     main()
{
        cout << "Please enter year, month: ";

        int     year, month, days = 0;

        cin >> year >> month;
```

```
    switch(month)       {
        case  12:   days += 31;
        case  11:   days += 30;
        case  10:   days += 31;
        case  9:    days += 30;
        case  8:    days += 31;
        case  7:    days += 31;
        case  6:    days += 30;
        case  5:    days += 31;
        case  4:    days += 30;
        case  3:    days += 31;
        case  2:
            if(year % 4 == 0 && year % 100 != 0 || year % 400 == 0)
                days += 29;                         // 闰年
            else
                days += 28;                         // 平年
        case  1:    days += 31;
    }
    if(days == 0)                                   // 输入的 month 与任何一个 case 都不匹配
        cout << "You entered a wrong month!" << endl;
    else
        cout << "The total days is " << days << endl;

    return 0;
}
```

设用户对该**程序**的响应为

Please enter year, month: **1998 11**<CR>

则**程序**的输出为

The total days is 334

即 1998 年中，从 1 月 1 日到 11 底的总天数为 334 天。

上例中，case 2 分支里 if **语句**的表达式是用来判断用户所输入的年份是不是闰年的。某年是闰年的条件是：该年年份能被 4 整除但不能被 100 整除，或者能被 400 整除。

例 3.6 的**程序**中，所有分支均无 break **语句**，与例 3.5 呈明显的对比。该**程序**的执行过程请读者对照例 3.5 自行分析。

由于 switch **语句**是合法的 C++**语句**，且在**程序**中起一条**语句**的作用，所以 switch **语句**中的任一 $statement_i$ 都可以是一条 switch **语句**，这种情况叫做 switch **语句**的嵌套。例如：

```
switch(a)   {
    switch(b)   {
        case  1:
            //...
    }
    case  1:
```

```
switch(a)   {
    case  1:
        switch(b)   {
            case  1:
                //...
        }
```

```
        //...                                    //...
    }                                      }
```

这时，内层 case 与外层 case 含有相同的常量值也不会出现矛盾，因为它们分别配匹不同的表达式。

3.2.5 if 语句和 switch 语句的比较

if 语句和 switch 语句都可以用来处理程序中的分支问题，在许多场合可以互相替代。但是它们之间还是有一些差别，其主要表现为：

1. if 语句常用于分支较少的场合；而 switch 语句常用于分支较多的场合。

2. if 语句可以用来判断一个值是否落在一个范围内；而 switch 语句则要求其相应分支的常量必须与某一值严格相等。例如，设 i 为一整型变量，若 i 的值为 1，2，3 或 4 时，就执某一操作 process，则相应的 if 语句和 switch 语句分别为：

```
if(i >= 1 && i <= 4)                    switch(i)    {
        process;                                case  1:
//...                                           case  2:
                                                case  3:
                                                case  4:
                                                        process;
                                        }
                                        //...
```

若值的范围较大时，显然 if 语句要优于 switch 语句。特别是当表达式的值是一个实数时，通常只能使用 if 语句。

3.3 循环语句

在一个程序中，常常需要在给定条件成立的情况下，重复地执行某些操作。C++语言为实现这一目的提供了 3 种循环语句：while 语句、do-while 语句和 for 语句。在循环语句中，重复执行的操作叫做循环体。循环体可以是单条语句、块语句甚至是空语句。

3.3.1 while 语句

while 语句具有以下的一般形式：

```
while(expression)
    statement;
```

该语句的执行过程为：若表达式 expression 为真，则反复执行其循环体 statement；若表达式的值为假，则不再执行 statement，转而执行循环体后边的语句。也就是说，当循环条件为真时则反复执行循环体。因此，while 语句也被叫做“当”型循环。

例 3.7 使用 while 语句。

```
//      EX3_7.CPP

#include      <iostream.h>
#include      <conio.h>                          // A

int     main()
{
        char    ch = 'a';

        cout << "Please enter some lower case letters, ";
        cout << "a period will terminates the program." << endl;

        while(ch != '.')        {
                ch = getch();
                if(ch == 'z' || ch == 'Z')
                        putch(ch - 25);
                else
                        putch(ch + 1);
        }

        return 0;
}
```

该程序提示用户输入一些小写字母，输入一个小数点将结束该程序。程序中的 getch（）和 putch（）都是 C++语言中的库函数，前者接受用户从键盘上输入的一个字符，在程序中可以用赋值运算符将该字符赋给一个字符型或整型变量；后者是将圆括号中表达式所代表的字符输出到屏幕上。程序中的行 A 是为能使用这两个函数所必需的。

由于该程序要重复地接受用户的输入，并将输入作一些调整后输出到屏幕上，所以采用了 while 语句来实现循环操作。while 语句中的表达式 ch != '.' 给定了循环的条件为字符型变量 ch 的值不等于小数点。由于在说明变量 ch 时，将它的值初始化为 'a'，这就防止了在程序进入 while 语句前，因为偶然因素使得 ch 的值恰巧为 '.'，因而导致循环一次都不执行。由于在循环体中 ch 用于接受用户从键盘上的输入，所以每循环一次，它的值都随用户的输入而在变化着。当它的值为 '.' 时，将使得表达式 ch != '.' 的值为假，从而导致循环结束。

运行该程序时，将会发现一个奇怪的现象：当你键入“a”时，屏幕的显示却是“b”；键入“b”，却显示“c”；……；键入“z”，却显示“a”。造成这一怪现象的原因有二：其一，库函数 getch（）是一个不回显的输入函数，即，它接受用户的输入，却并不将用户键入的内容显示在屏幕上。充分利用该函数的这一特性，可以将程序设计得非常有趣；其二，程序将用户输入的内容除“z”以外，均将其值加一以后输出。第 2 章中曾介绍过，char 型变量用于存储字符，也可以存储 8 位的二进制数。事实上，字符也是以 8 位的二进制数的形式存储的，其值就是该字符的 ASCII 码值，所以可以对其进行算术运算。由附录可知，字母字符在 ASCII 码表中是连续排放的，所以一个字母的 ASCII 码值加一就是其后继字母的 ASCII 码值。比如，字母“a”的 ASCII 码值为十进制的 97，字母“b”的 ASCII 码值为 98，等等。若用户键入的是字母“z”，它的 ASCII 码值为 122，122 减去 26 等于 97，正好是字母“a”的 ASCII 码值。应当说明的是：在对该程序进行输入时，只能键入字母，若键入其它字符则可能显示一些乱字符。

3.3.2 do-while 语句

do-while 语句具有以下的一般形式：

```
do  {
    statement;
} while(expression);
```

do-while 语句与 while 语句的执行过程基本相同，所不同的是：while 语句先判断给定条件是否成立，后执行循环体；而 do-while 语句则正好相反——先执行循环体后判断表达式 expression 的值，若表达式的值为真则反复地执行循环体，直到表达式的值为假才退出循环。因此，do-while 语句也被叫做“直到”型循环。在一定条件下，while 循环可能一次都不执行，而 do-while 循环在任何条件下都至少要执行一次。这一点正是在程序设计中决定选择 while 语句还是 do-while 语句的重要依据。

例 3.8 例 3.7 的另一版本。

```
//      EX3_8.CPP

#include      <iostream.h>
#include      <conio.h>

int     main()
{
        char    ch;

        cout << "Please enter some lower case letters, ";
        cout << "a period will terminaes the program." << endl;
        do      {
                ch = getch();
                if(ch == 'z' || ch == 'Z')
                        putch(ch - 25);
                else
                        putch(ch + 1);
        } while(ch != '.');

        return 0;
}
```

比较例 3.7 和例 3.8 可以发现，两者除了循环语句不同外，最主要的区别在于前者需要对变量 ch 进行初始化，而后者则不必这样做。

应说明的是：do-while 语句中的循环体若是一个单条语句或空语句，则可以省略其中的一对花括号。但为了提高程序的可读性，习惯上在任何条件下都保留这一对花括号。另外，由于整个 do-while 是一条语句，所以 while 后边的分号“;”不得省略。

3.3.3 for 语句

for 语句是 C++语言中最为灵活的循环语句，可以毫不夸张地讲：C++语言中的 for 语句

完全可以替代 while 语句和 do-while 语句。

3.3.3.1 for 语句的基本形式

for 语句具有以下的一般形式：

for(initialization; condition; increment)
***   statement;***

其中：initialization，condition 和 increment 均为任一合法的 C++表达式。initialization 叫做初始化部分，它通常用来设置循环控制变量（简称循环变量）的初值；condition 叫做循环条件部分，它用来设置循环的条件；increment 叫做增量部分，它通常用来修改循环变量的值。

for 循环的执行过程为：当 condition 为真时，就反复执行循环体；否则终止循环，执行 for 语句的后继语句。每循环一次，for 语句就根据 increment 表达式的规定修改循环变量的值。

在程序中，for 循环的基本用法为：先说明一个整型或字符型变量作为循环变量；然后在初始化部分为循环变量置一初值、在循环条件部分用一关系表达式给出当循环变量的值在何范围时继续循环、在增量部分用一赋值表达式给出循环变量的变化量；最后，给出循环体。

例 3.9 输出 1 到 10 的平方。

```
//      EX3_9.CPP

#include      <iostream.h>

int     main()
{
        int   j = 1;

        for(int i = 1; i <= 10; i ++)    {
                cout << (j * j) << "    ";
                j ++;
        }
        cout << endl;                                   // A

        return 0;
}
```

该程序的输出为：

```
1   4   9   16   25   36   49   64   81   100
```

本例中，循环变量 i 的初值为 1，并规定当 i 的值小于等于 10 时重复执行循环体，且每循环一次，循环变量的值加 1。当 i 的值大于 10 时，终止循环，执行语句 A。

循环变量不仅用在 for 语句的控制部分，而且也可以在循环体中使用，甚至可以在循环体中修改循环变量的值。

例 3.10 在循环体中修改循环变量的值。

```
//      EX3_10.CPP
```

```
#include      <iostream.h>

int     main()
{
        for(int i = 1; i <= 10; i ++)
                cout << (i *= i) << "    ";
        cout << endl;

        return 0;
}
```

该程序的输出为：

```
1       4       25
```

从该程序的输出可以看出，循环体仅被执行了 3 次，其原因就在于循环体中修改了循环变量的值：第一次执行循环体时，循环变量 i 的值为 1，而循环体中的 i *= i 实质上并未改变它的值；第二次执行循环体时，循环变量 i 的值为 2，而上述表达式则将 i 的值修改为 4；这就导致第三次执行循环体时 i 的值为 5，但上述表达式又将它修改为 25，从而使得循环变量的值超出了循环条件所限定的范围，循环因此而终止。

应当强调指出的是：在循体中错误地修改循环变量的值将可能导致不可预知的后果。比如，例 3.11 就是一个死循环的例子。

例 3.11 一个死循环。

```
//      EX3_11.CPP

#include      <iostream.h>

int     main()
{
        for(int i = 0; i < 2; i ++)       {
                cout << "This program will run forevermore\n";
                i --;
        }

        return 0;
}
```

该例中，循环变量 i 的增量为 1，但是，每循环一次循环体中又将 i 的值减去 1，从而使得其值永远为 0。这样将导致循环条件 i < 2 永远成立。

3.3.3.2 含有逗号表达式的 for 语句

一般情况下，for 语句中的初始化部分和增量部分总是一个赋值表达式；循环条件部分总是一个关系表达式或逻辑表达式。但在实用中，这些部分常常使用逗号表达式为其设置多个循环控制变量，或用多个条件对循环进行控制。

例 3.12 含有逗号表达式的 for 语句。

```
//      EX3_12.CPP
```

```
#include      <iostream.h>

int     main()
{
        int     i, j;

        for(i = 1, j = 1; i <= 10, j < 30; i ++, j ++)   {
                cout << (i * j) << "   ";
                j += 3;                                          // A
        }
        cout << endl;

        return 0;
}
```

该程序的输出为：

1 10 27 52 85 126 175 232

该例中的 3 个部分均用的是逗号表达式，为循环设置了两个循环控制变量 i 和 j，且循环条件也为两个：i <= 10 和 j < 30（在本例中，该逗号表达式可以用逻辑表达式 i < 10 || j < 30 来代替）。每循环 1 次 i 的值加 1；j 的值加 4。当循环了 9 次以后，j 的值已超过了 30，从而导致循环的结束。

并非必须将 for 语句中的 3 个表达式都对应地写成逗号表达式，上例完全可以改写成下例的形式而功能不变。

例 3.13　例 3.12 的另一版本。

```
//      EX3_13.CPP

#include      <iostream.h>

int     main()
{
        int   i, j = 1;

        for(i = 1; i <= 10, j < 30; i ++, j += 4)
                cout << (i * j) << "   ";
        cout << endl;

        return 0;
}
```

3.3.3.3　for 语句的省略

一般说来，for 语句中的 3 个表达式部分并无必然的联系，在实用中，根据实际需要可以省略其中的任何一部分，乃至全部。省略时，必须保留分号，以便使编译器知道省略的是哪一部分。例如：

```
for(;;)  {
     //...
}
```

就是一个合法的 for 循环。对于这种循环，循环体中必须有使循环终止的机构，否则，它将是一个死循环，没有任何实用价值。例 3.14 就是一个有省略的循环。

例 3.14 例 3.7 的又一版本。

```
//      EX3_14.CPP

#include      <iostream.h>
#include      <conio.h>

int     main()
{
        char    ch;                                             // A

        cout << "Please enter some lower case letters, ";
        cout << "a period will terminates the program." << endl;

        for(ch = getch(); ch != '.');   {                       // B
                if(ch == 'z')
                        putch(ch - 25);
                else
                        putch(ch + 1);

                ch = getch();
        }

        return 0;
}
```

该程序与例 3.7 中程序的功能完全相同。即该循环可能一次都不执行，若用户第一次就输入一个“.”的话。

如果将例中的语句 A 改为：char ch = 'a'（或任一小写字母）；将行 B 改为：for(; ch != '.');，其功能将与例 3.8 中程序的功能完全相同。由此可见，由于 for 语句具有极大的灵活性，它完全可以取代 while 语句和 do-while 语句。当然，在许多场合下，使用 while 或 do-while 语句显得更自然些。

3.3.4 三种循环语句的比较

虽然 C++语言中的 for 语句极为灵活，但在实用中，为了提高程序的可读性，使程序中的语句显得自然、简洁，仍广泛地使用着 while 和 do-while 语句。

一般讲，若循环次数已知，则选用 for 语句，否则选用 while 或 do-while 语句。对于后者，若无论在什么条件下，循环体至少得执行一次，则应选用 do-while 语句；否则选用 while 语句。

3.3.5 循环的嵌套

循环语句中，无论其循环体是单条语句、复合语句还是空语句，整个循环语句在语法上起一条语句的作用。因此，循环语句可以出现在程序中任何需要一条语句的地方。当然也可

以作为循环体中的语句，这样就形成了所谓循环的嵌套。

例 3.15 打印九九表。

```
//      EX3_15.CPP

#include      <iostream.h>
#include      <iomanip.h>                                    // C

int     main()
{
        int     i, j;

        for(i = 1; i < 10; i ++)
                cout << setw(4) << i;                        // A
        cout << endl << endl;

        for(i = 1; i < 10; i ++) {
                for(j = 1; j < 10; j ++)
                        cout << setw(4) << (i * j);          // B
                cout << endl;
        }

        return 0;
}
```

该程序的输出为：

```
   1   2   3   4   5   6   7   8   9

   1   2   3   4   5   6   7   8   9
   2   4   6   8  10  12  14  16  18
   3   6   9  12  15  18  21  24  27
   4   8  12  16  20  24  28  32  36
   5  10  15  20  25  30  35  40  45
   6  12  28  24  30  36  42  48  54
   7  14  21  28  35  42  49  56  63
   8  16  24  32  40  48  56  64  72
   9  18  27  36  45  54  63  72  81
```

程序里，行 A 和行 B 中 setw(4)的作用是要求将输出数据按 4 个字符的宽度输出。这里，setw（）是 C++语言 I / O 流类 ios 的一个预定义的输出操纵符，程序中的行 C 是为能使用该操纵符所必须的。关于 C++的 I / O 流类将在第 13 章中介绍。

3.4 循环的中断

当循环语句中的循环条件不满足时，就会结束循环体的执行。然而在实用中，常常会由于某些原因需要，在循环条件仍旧满足的情况下，终止某次循环乃至整个循环。C++语言为

此目的提供了中断循环的手段。

3.4.1 break 语句

前面介绍 switch 语句时，曾使用过 break 语句，它的作用是使某个分支的执行到此结束。除此之外，break 语句还可以用在循环语句中，它的执行将导致循环语句的结束。

例 3.16 用 break 语句终止循环。

```
//      EX3_16.CPP

#include    <iostream.h>
#include    <conio.h>                                   // A

int     main()
{
        while(1)        {
                cout << "This loop will run forever, until you hit a key.\n";

                if(kbhit())
                        break;
        }
        cout << "\nLoop is over.\n";

        return 0;
}
```

该程序中 while 语句的表达式为 1，即循环条件永远成立。所以，该程序一经运行，就不停地在屏幕上显示：

This loop will run forever, until you hit a key.

直到用户按下任何一个键，上述显示才停止，屏幕上将显示：

Loop is over.

且程序的运行到此结束。

程序中的 kbhit（）是 C++语言的一个库函数，它的功能是检查是否有键被按下。若有键按下，该函数就返回 1（真），否则返回 0（假）。程序中的行 A 是为能使用该函数所必须的，（注意：该函数仅能用于 DOS 系统）。若有键按下，则 if 语句中的表达式就为真，使得 break 语句得以执行，从而导致 for 循环的结束。

3.4.2 continue 语句

continue 语句的唯一作用就是中止一趟循环的执行。它使得本趟循环中尚未执行的语句不再执行，紧接着执行下一轮的循环。

例 3.17 使用 continue 语句。

```
//      EX3_17.CPP
```

```
#include      <iostream.h>
#include      <stdlib.h>                          // A

int     main()
{
        for(int i = 0; i < 100; i ++)    {
                int     n;

                n = rand() % 100;                   // B
                if(n % 2)
                        continue;
                cout << n << endl;                  // C
        }

        return 0;
}
```

该**程序**的作用是输出不超过 100 个值为 0～99 的随机偶数。

程序中的 rand（）是 C++**语言**的库函数，它的功能是产生一个其值为 0～2^{15} - 1 的随机整数。行 A 为能使用该函数所必须的。

程序中的 for 循环最多执行 100 次。循环体中的**语句** B 通过将产生的随机数模除 100，使 n 的值为一 0～99 的随机数。若 n 的值为一奇数，则 if **语句**中表达式的值为真（表达式 n % 2 在这时等效于 n % 2 != 0），使得 continue **语句**得以执行，从而在本次循环中就不执行循环体中的输出**语句** C；否则就输出 n 的值。

3.4.3 exit（）函数和 abort（）函数

exit（）和 abort（）函数是 C++语言中的两个库函数。为能在**程序**中使用这两个函数，必须在**程序**的首部加上一行：

```
#include <stdlib.h>
```

exit（）和 abort（）函数的作用都是终止整个**程序**的运行，将**控制**返还给操作系统。所不同的是，前者在结束**程序**的运行前，还会做一些善后工作。比如，将**程序**中的变量、对象等所占的存储单元释放掉；关闭所有已打开的文件等。而后者仅仅结束**程序**的运行，不做任何善后工作。

这两个函数在使用时，均要求在圆括号中放一个常数来表示退出的原因。习惯上用 0 表示正常退出，而非零表示非正常退出。

例 3.18 使用 exit（）函数。

```
//      EX3_18.CPP

#include      <iostream.h>
#include      <stdlib.h>

int     main()
{
```

```
    int     sum = 0, n = 0, m;

    cout << "Enter some integers, zero will terminate the program. \n";
    do      {
            cin >> m;
            sum += m;
            n ++;
            if(m == 0)      {
                    cout << "You have entered " << n << " integers.";
                    cout << "Theirs sum is " << sum << endl;

                    exit(0);
            }
    } while(1);

    return 0;
}
```

该程序要求用户输入一些整数，当输入的数为 0 时，程序输出用户所输入数的个数以及它们的总和，并结束运行。

3.4.4 补充说明

以上介绍的 3 种中断循环的手段中，continue 语句的使用范围最小，它只能用于循环语句中，且仅能中止一趟循环的继续执行。break 语句不仅可以用来结束整个循环的执行，而且更多地被用在 switch 分支语句中。而 exit（）和 abort（） 函数则可以放在程序中的任何地方，以终止整个程序的执行。

continue 和 break 语句的作用范围仅为一层循环，对于有嵌套的循环，它们只对其所在的那层循环起作用。若欲结束一个多层的循环嵌套，则必须使用多个 break 语句。

例 3.19 用 break 语句退出多重循环。

```
//      EX3_19.CPP

#include      <iostream.h>
#include      <conio.h>

int     main()
{
        int     n, done;

        n = done = 0;
        cout << "Looping..., press any key to stop.\n";
        while(1)        {
                while(1)        {
                        while(1)        {
                                if(kbhit())     {
                                        done = 1;

                                        break;
```

```
                }
                n++;
            }
            if(done)
                break;
        }
        if(done)
            break;
    }
    cout << "The loop have run " << n << " times.\n";

    return 0;
}
```

该程序首先在屏幕上显示：

Looping..., press any key to stop.

然后进入一个三重的死循环，直到用户按下任一键，程序将显示最内层循环执行的次数并结束运行。

在程序运行的过程中，若用户按下任意一个键，最里层的循环就将退出标志 done 置 1，然后执行 break 语句退出最内层循环；否则就将 n 加 1。在第二、第一层循环中，循环体测试退出标志 done 的值是否为 1，以决定是否执行 break 语句退出本层循环。由于 break 语句只能终止本层循环的执行，所以设置标志以供测试在这里是必须的。

3.5 goto 语句与标号

goto 语句是许多程序设计语言中均具有的程序控制语句。尤其是在早期的 BASIC 语言中，使用 goto 语句几乎成了控制程序流程的基本手段。但大量的程序设计实践表明，在程序中滥用 goto 语句会破坏程序的逻辑结构，使得程序难于阅读、难于调试。因此，计算机工作者中对 goto 语句多有微词，以致有人甚至提出要求在程序设计语言中取消 goto 语句。但另一方面，在有些场合中，适当地使用 goto 语句却大大地提高程序的可读性，改善了程序的运行效率。

例 3.20 例 3.19 程序的另一版本。

```
//      EX3_20.CPP

#include      <iostream.h>
#include      <conio.h>

int     main()
{
        int     n = 0;

        cout << "Looping..., press any key to stop.\n";
        while(1)
                while(1)
```

```
        while(1)        {
                if(kbhit())
                        goto stop;
                n ++;
        }

    stop:                                   // A
    cout << "The loop have run " << n << " times.\n";

    return 0;
}
```

在本例运行过程中，如果用户按下了任何一个键，将导致 goto 语句的执行，从而使程序的执行跳出所有循环，转而执行 stop:（行 A）后边的语句。与例 3.19 相比较，本程序显得更加简洁易读。另外，由于本例少了一个变量、两个判断和两条 break 语句，该程序的时间、空间开销均小于上例。

goto 语句的一般形式为：

goto label;
//...
label: statements;

其中：label：是一个标号，而 label 是标号的名字。在 C++语言中，标号的命名应符合标识符命名的要求。

goto 语句的作用是使程序的执行无条件地转向 label：后边的语句。

应强调指出的是：可以利用 goto 语句从循环中跳出，但不得用 goto 语句从循环体外跳转到循环体中去。

goto 语句不仅可以用在循环语句中，也可以用在程序中任何需要它的地方（尽管在现代程序设计中很少有这种需要）；goto 语句可以出现在标号前边（称向前跳转），也可以出现在标号后边（称向后跳转）。

尽管 C++语言中提供了 goto 语句，但它决不鼓励程序员使用 goto 语句。甚至可以这样来看待 goto 语句：在早期的程序设计方法中，goto 语句是控制程序流程的基本手段之一；而在现代程序设计方法中，它则变成了程序设计高手的技巧之一。

习题

3-1 将本章给出的例题输入计算机，编译并运行它们。

3-2 试将例 3.3 中的 if 语句改为 switch 语句。

3-3 试将例 3.14 中的 for 循环分别改为 while 循环和 do-while 循环。

3-4 将第 2 章中的习题 2-5，2-6 和 2-7 分别改循环的形式，以便它们可以反复地接受用户的输入并计算出相应的结果。当用户键入某一指定值时结束程序的运行。

3-5 编写一个程序，要求用户输入一个不超过 5 位的整数，计算并输出该数的各位之和。比如，若用户输入的整数为 1998，则输出应为 27。

3-6 编写一个程序，将用户输入的百分制成绩转换成对应的五级计分制成绩并输出。90 分

以上为 A；80～89 分为 B；70～79 分为 C；60～69 分为 D；60 分以下为 E。

3-7 编写程序打印所有的出水仙花数。所谓“水仙花数”是指一个 3 位数，其各位数字的立方和等于该数本身。例如，$153 = 1^3 + 3^3 + 5^3$，所以 153 是一个水仙花数。

3-8 编写程序输出 1 000 以内所有的完数。所谓“完数”是指与其因子之和相等的数。例如 $6 = 1 + 2 + 3$，而 1，2 和 3 均是 6 的因子。该程序要求以如下的形式输出：

6-->1, 2, 3

3-9 编写程序输出 100 以内的 Fibonacci（裴波那契）数列。所谓“Fibonacci” 数列是指形式如：

$a_0 = 1$，$a_1 = 1$，...，$a_n = a_{n-1} + a_{n-2}$，...

的数列。

第4章　函数

函数是C++程序的构成基础。一个C++程序就是由一个个函数所组成的，即便是最简单的程序，也得有一个 main（）函数。因此，一个 C++程序无论多么复杂，规模有多么大，程序的设计最终都落实到一个个函数的设计和编写上。

4.1　函数

函数是一个命名的程序代码块，是程序完成其操作的场所。在程序设计中，有许多算法是通用的，例如求一个数的平方根，解一元二次方程。如果将这些算法定义为函数，程序员就可以在程序中需要这些算法的地方直接使用它们，而不必再作定义。

4.1.1　定义函数

一个函数必须先定义才可以使用，所谓定义函数就是编写完成函数功能的程序块。定义函数的一般格式为：

```
<type>      func_name(<arg_list>)
{
      func_body
}
```

其中：type 为函数返回值的类型，它可以是任一基本数据类型或导出数据类型；func_name 是用户为函数所起的名字，它应遵守标识符命名的规定；arg_list 是函数的参数表（下节介绍）；而 func_body 为函数体，即函数所完成的具体操作，它由一系列说明和语句组成。比如，本章以前所有例子中给出的 main（）函数就属于函数定义。

例4.1　定义一个阶乘函数。

```
//      EX4_1.CPP

#include      <iostream.h>

void   facto()
{
       cout << "Enter a integer: ";

       long   fac = 1;
       int    n;

       cin >> n;
```

```
    for(int i = n; i > 0; i --)
        fac *= i;
    cout << "The factorial of " << n << " is " << fac << endl;
}
```

该函数定义了一个阶乘的算法。该函数一经定义，就可以在程序中多次地使用它。函数的使用是通过函数调用来实现的。

4.1.2 函数的调用

在 C++程序中，除了 main（）函数以外，任何一个函数都不能独立地在一个程序中存在。任一函数的执行都是通过在 main（）函数中直接或间接地调用该函数来引发的。调用一个函数就是执行该函数之函数体的过程。

函数调用的一般形式为

func_name(<args>);

其中：args 是与函数定义时所给出的参数表相对应的数据（下节介绍）。

例 4.2 阶乘函数的调用。

```
//      EX4_2.CPP

#include      <iostream.h>

void    facto()
{
        cout << "Enter a integer: ";

        long    fac = 1;
        int     n;

        cin >> n;
        for(int i = n; i > 0; i --)
                fac *= i;
        cout << "The factorial of " << n << " is " << fac << endl;
}

int     main()
{
        facto();

        return 0;
}
```

该例通过编译、连接，可以生成一个可执行的程序。运行该程序就可以实现对 facto（）函数的调用。对本例来讲，程序实际上只是执行了 facto（）函数的功能。

4.2 函数间的数据传递

在 C++程序中，利用函数不仅可以改善程序的可读性，而且提高了程序的灵活性。但上

节中所介绍的 facto（）函数却没有多大的实用价值。首先，它是通过交互的方式要求用户输入被计算的数值，而程序本身却无法设置该值；其次，函数中通过计算得到的数据被函数直接送到屏幕上显示，而程序本身却无法使用该数据。

实际上，一个函数可以从函数之外获得一些数据，并可向其调用者（在 C++中叫做主调函数或调主）返回一些数据。这些数据主要是通过函数的参数与函数的返回值来传递的。

4.2.1 函数的参数和返回值

一个函数利用参数从主调函数中接受数据，并通过返回值向主调函数返回数据。

4.2.1.1 函数的参数

一个函数可以有零个或多个参数，这些参数必须在函数定义时在其参数表中进行说明，说明的形式与变量说明类似。例如：

```
f(int i, int j, char k)
{
    //...
}
```

说明了一个带有 3 个参数的函数 f（），它们分别是整型参数 i，j 和字符型参数 k。

在说明函数的参数时，每个参数的数据类型都必须单独给出，例如，上述的参数说明不得写成

```
f(int i, j, char k)
```

在参数表中说明的变量叫做函数的形式参数，简称形参。在调用函数时，一般须为每一个形参给出其实际数据——实际参数，简称实参。实参可以是一个变量、一个常量、一个表达式、甚至可以是一个函数调用，但每一个实参的数据类型应该与其所对应的形参的数据类型相匹配。例如，设有变量说明：

```
int    m, n;
char   c;
```

则以下均为对函数 f（）的合法调用：

```
f(m, n, c);
f(n, m, c);
f(m, 5, 'a');
```

而以下的函数调用则会引起编译器的警告：

```
f(c, m, n);
f('a', m, 30);
f(m, n);
```

其中，前两个函数调用中实参与形参的类型不匹配，最后一个函数调用中实参的个数与形参不匹配。

由于 C++语言是一种强类型检查语言，C++编译器会检查实参与形参结合时，其个数及

类型的匹配情况。

若在函数定义中参数表为空，即一个形参都没有，则 C++编译器将该函数解释成可能带有任意多个参数。这种写法实际上是要求 C++编译器不要对参数的匹配情况作检查。若欲明确说明某函数一个参数都不带，则应使用 void 关键字。例如：

```
f(void)
{
    //...
}
```

就说明函数 f（）一个参数都没有，因而在调用它时也不应当带有实参，如例 4.1 所示。

4.2.1.2 函数的返回值

一个函数可以有零个或一个返回值，称为函数的值，在函数定义时应给出其返回值的类型。例如：

```
int   f(char c)          // A
{
    //...
}
```

就定义了一个值为整型数、且带有一个字符型参数的函数 f（）。

若省略了函数值的类型，则 C++编译器将缺省地将它当作值为 int 型的函数处理。例如，上例的行 A 可改写为

```
f(char c)
```

若欲明确地说明某函数无返回值，则应使用 void 关键字。例如：

```
void  f(void)
{
    //...
}
```

就定义了一个既无参数又无返回值的函数 f（）。

应指出的是：在 C++语言中，允许对函数的返回值不予理会。例如，下面的函数调用均是合法的：

```
int   f(void)
{
    //...
}

int   i = f();           // 将函数 f（）的返回值赋给整型为量 i
f();                     // 忽略函数 f（）的返回值
```

4.2.1.3 return 语句

关键字 return 可以在函数中构成 return 语句。return 语句在程序中有两个作用：其一是

终止函数的执行，返回到主调函数；其二是不仅终止函数的执行，而且还向主调函数返回函数的值。return 语句的一般形式为

return <expression>;

其中：expression 是一个可选的且其值与函数值类型相同的表达式。若无此项，则 return 语句起其第一种作用；否则起其第二种作用。

注意：在实用中常常有人将 return 后边的表达式用一对圆括号括起来，这与不带括号没有什么区别，例如：

return a + b * c;　　和　　return(a + b * c);

两条语句的作用完全相同。若没有使用括号，则表达式与 return 之间至少应当有一个空白字符。

例 4.3 阶乘函数的另一版本。

```
//      EX4_3.CPP

#include      <iostream.h>

unsigned long        facto(int n)
{
      unsigned long        fac = 1;

      for(; n > 0; -- n)
            fac *= n;

      return fac;
}

int     main()
{
      unsigned long        num;

      num = facto(10);
      cout << "The factorial of 10 is " << num << endl;

      int     m = 8;

      cout << "The factorial of " << m << " is " << facto(m) << endl;

      return 0;
}
```

该程序的输出为：

```
The factorial of 10 is 3628800
The factorial of 8 is 40320
```

由本例可见，有值的函数可以出现在程序中任何一个需要值的地方。

4.2.2 函数原型

函数的定义属于对函数的定义性说明。C++语言允许并鼓励对函数进行引用性说明，而且次数不限。在 C++语言中，对函数的引用性说明就叫做函数原型（Function Prototype），通常也称其为函数说明。函数原型的一般形式为

<type>　　func_name(<arg_list>);

其中各部分的意义与函数定义相同。

函数原型与函数定义的区别在于：函数原型没有函数体部分，且是用分号结束的，就像变量的说明（注意：变量说明通常属于定义性说明）。

在例 4.2 和例 4.3 中，函数 facto（）均出现在对它的调用之前。在那里必须这样做，否则将会引起一个编译错误而导致编译失败。若使用了函数原型，并且只要将函数原型放在程序中对函数第一次调用之前，就可以将函数的定义放在对它的调用之后，甚至可以放在其他文件中（参见本章中“程序的多文件组织”）。这就为进行结构化、模块化程序设计提供了极大的方便。

例 4.4　使用函数原型。

```
//      EX4_4.CPP

#include      <iostream.h>

int     main()
{
        int     max(int x, int y);                              // A

        cout << "Please enter two integer: ";

        int     a, b;

        cin >> a >> b;
        cout << "The maximum is " << max(a, b) << endl;         // B

        return 0;
}

int     max(int x, int y)
{
        return (x >= y ? x : y);
}
```

本例中，尽管函数 max（）的定义出现在对它的调用（程序中行 B）之后，然而由于使用了函数原型（程序中的行 A），程序就能成功地编译通过。这是由于编译器在编译行 B 时已经在行 A“见到过”max 这个“名字”了，并且“知道”它是一个函数以及该函数的参数与返回值的情况。从而能够对行 B 中的调用进行类型检查。

应当说明的是：不得将函数 max（）的定义来替换程序中的行 A。这是因为在 C++语言中，不允许在函数体中定义函数。

函数原型向编译器提供了函数的名字、值的类型和参数的个数及类型等信息。在函数原型中，参数的名字属于引用性说明的，因此，上例中的行 A 可以改写为

```
int     max(int, int);
```

即，在函数原型中，只需指明参数的个数以及各自的类型即可（关于这一点，在 4.4 节中还会有进一步地讨论）。

4.2.3 函数的值调用

在调用一个带有参数的函数时，就存在一个实参与形参结合方式的问题。在 C++语言中，实参与形参有 3 种结合方式：传值调用、传地址调用和引用调用。这里介绍传值调用，其余两种结合方式将在第 6 章中介绍。

传值调用简称为值调用。值调用时，实参仅将其值赋给了形参，因此，在函数中对形参值的任何修改都不会影响到实参的值。

例 4.5 值调用时实参的变化。

```
//      EX4_5.CPP

#include      <iostream.h>

int     cube(int);

int     main()
{
        int     n = 5, m;

        m = cube(n);
        cout << "The cube of " << n << " is " << m << endl;
        cout << "In main(), the n is still " << n << endl;

        return 0;
}

int     cube(int x)
{
        x = x * x * x;
        cout << "In cube(), the argument x now is " << x << endl;

        return x;
}
```

该程序运行的结果为：

```
In cube() the argument x now is 125
The cube of 5 is 125
In main(), the n is still 5
```

可见，尽管在函数 cube（）中，形参 x 的值被改变了，而实参 n 的值仍旧保持不变。

值调用的好处是减少了主调函数与被调函数之间的数据依赖，增强了函数自身的独立性。当然，在实际应用中，有时需要用函数来修改实参的值。为此目的，就得使用第 6 章将要介绍的地址调用和引用调用。

4.3 C++的库函数

为了方便用户的使用，任何一个 C++软件包都提供了数百个已预定义了的函数，这些函数就叫做 C++的库函数，或叫系统函数、标准函数。对于库函数，用户不用定义也不用说明就可直接使用。由于 C++软件包将不同功能库函数的函数原型分别写在不同的头文件中，所以，用户在使用某一库函数前，必须用 include 预处理指令给出该函数的原型所在头文件的文件名。例如，欲使用库函数 sqrt（），由于该函数的原型在头文件 math.h 中，所以必须在程序中对该函数调用之前书写一行：

```
#include <math.h>
```

由于 C++语言的库函数常常多达四五百个，且不同实现中的库函数在某些方面存在着很大的差异，所以本书不可能将它们一一介绍。在这里，仅给出在以前章节出现过的库函数的函数原型以及函数原型所在头文件的文件名。在后继章节中，还会陆续给出一些 C++的库函数并说明它们的用法。

注意：本书给出的所有库函数以及与它们有关的头文件，都是来自 Borland C++软件包。若读者使用的是其它的 C++编译器，则可能有些头文件的文件名有所不同，但函数原型却是完全相同的。这是因为本书所给出的库函数均是各种 C++标准所规定的库函数。

abort（）
函数原型：void abort(void);
头文件：stdlib.h

exit（）
函数原型：void exit(int);
头文件：stdlib.h

fabs（）
函数原型：double fabs(double);
头文件：math.h

getch（）
函数原型：int getch(void);
头文件：conio.h

kbhit（）
函数原型：int kbhit(void);
头文件：conio.h

putch（）
函数原型：int putch(int);
头文件：conio.h

rand（）
函数原型：int rand(void);
头文件：stdlib.h

sqrt（）
函数原型：double sqrt(double);
头文件：math.h

C++库函数所涉及的操作非常广泛，数量又极为庞大，所以程序员应常备一本 C++库函数手册，以便随时查阅。当然，对于英文较好的人，可以直接查阅所使用的 C++软件包的联机帮助。

4.4 作用域和存储类

C++语言主要是用来编写大型软件的，而一个大型软件往往是由多人合作完成的，这就不可避免地会出现标识符重名的问题。C++语言的作用域规则和存储类说明就是解决这个问题的有效手段。

4.4.1 作用域

作用域体现了标识符的可见性。一个标识符的作用域是程序中的一块区域，在这块区域中，该标识符是可见的，即可访问的。出了这个区域，该标识符则是不可见的，即不可访问的。

C++语言的作用域共有 5 种：块作用域、文件作用域、函数原型作用域、函数作用域和类作用域。类作用域将在第 8 章中介绍，这里介绍其他 4 种作用域。

4.4.1.1 块作用域

块是程序中被一对花括号括起来的一块区域。在块内说明的标识符具有块作用域，它开始于其说明处，结束于块的结尾处。具有块作用域的标识符称作局部标识符。例如：

```
void  f(void)
{
      {
            int    i;

            i = 5;              // 正确，变量 i 可访问
      }
      cout << i << endl;        // 错误，i 不可访问
}
```

对于块作用域，这里应该说明 3 点：

1. 具有块作用域的标识符在其作用域内将屏蔽其作用域包含本块的同名标识符。

例 4.6 具有块作用域的标识符。

```
//      EX4_6.CPP

#include      <iostream.h>

int     main()
{
      int     k;

      k = 10;
      for(int i = 0; i < 5; ++ i)      {
            int     k;

            k = i + 1;
            cout << k << '\t';                  // A
```

```
    }
    cout << endl;
    cout << k << endl;                    // B

    return 0;
}
```

该程序的输出为：

```
1    2    3    4    5
10
```

由程序的输出可见，语句 A 输出的是 for 语句中说明的局部变量 k，而语句 B 输出的是在 main（）函数说明的变量 k（也是一个局部变量）。在本例中，尽管在 main（）函数中说明的变量 k 之作用域包含了 for 语句的语句段，但由于在 for 语句中说明的同名变量对它的屏蔽作用，使得它在整个 for 语句中暂时不可见。

2. 出现在一个函数定义中的函数原型使得与该函数原型所对应的函数具有块作用域，即所说明的函数只能被说明它的函数所调用。例如：

```
void  f();

int   main()
{
      f();                    // 正确，函数 f()具有文件作用域
      g();                    // 错误，函数 g()这里不可访问
}

void  f(void)
{
      void  g(void);

      g();                    // 正确，函数 g()具有块作用域
}

void  g(void)
{
      //...
}
```

3. 在 for 语句中说明的循环控制变量具有块作用域，其作用域仅为 for 语句本身。例如：

```
//...
for(int i = 0; i < 20; i ++)
    ;                                 // 空循环，什么都不做
cout << i << endl;                    // A 非法，变量 i 不再存在
```

注意：这一条与具体实现有关。在有些具体实现中，其作用域为包含 for 语句的块。也就是说，对于有些 C++编译器来讲，上述的行 A 是合法的。因此，为了保证程序的移植性，

在程序中尽量不要在 for 语句之外使用 for 语句中所说明的循环变量。

4.4.1.2 文件作用域

在所有的块以外说明的标识符具有文件作用域。具有文件作用域的标识符称为全局标识符，它整个文件中都是可以访问的。与块作用域标识符相同，具有局部作用域的同名标识符在其作用域中将暂时屏蔽全局标识符。然而，与块作用域标识符不同的是，在一个局部区域中，可以用作用域运算符“::”来访问具有文件作用域的标识符。

例 4.7 在局部域中访问被屏蔽了的文件作用域标识符。

```
//      EX4_7.CPP

#include      <iostream.h>

int     k;

int     main()
{
        k = 10;
        for(int i = 0; i < 5; ++ i)      {
                int     k;

                k = i + 1;
                cout << k << '\t';
                :: k ++;                                    // 访问全局变量 k
        }
        cout << endl;
        cout << k << endl;

        return 0;
}
```

该程序的输出为：

```
1     2     3     4     5
15
```

程序中，for 循环每循环一次就将全局变量 k 加 1，所以，最终输出全局变量 k 的值为 15。

4.4.1.3 函数原型作用域

在函数原型中说明的标识符具有函数原型作用域，它起始于其说明处，结束于函数原型的末尾处。正因为如此，参数的名字在数原型中可以与函数定义中不同。例如：

```
int     f(int m, int n);                // 函数原型

int     f(int x, int y)                 // 函数定义
{
        //...
}
```

正是由于在函数原型中所说明的标识符具有函数原型作用域，所以在函数原型中，参数

名可以省略掉。

4.4.1.4 函数作用域

具有函数作用域的标识符在说明它的函数中的任何地方均为可见的。在 C++语言中，只有标号具有函数作用域。标号是用来标明 goto 语句跳转的位置的，每个标号的名字在函数中必须是唯一的。例如，下面的函数定义中由于说明了两个同名的标号 lab，因而将导致一个编译错误：

```
void  f(int x)
{
      lab:
      cout << "A label" << endl;
      {
            lab:
            cout << "Another label" << endl;      // 错误，标号 lab 被重复说明
            //...
      }
      if(x < 5)
            goto lab;
      //...
}
```

若函数中确实需要多个标号，以便根据不同的条件使程序的执行转移到不同的程序段去，则在同一个函数中每一个标号都应该具有不同的名字。正是由于标号具有函数作用域，所以不可能用 goto 语句从一个函数转移到另外一个函数中。

4.4.2 存储类

存储类主要是针对变量而言的，它体现了变量的生存期，即变量占有内存空间的期限。变量的存储类同时也影响着 C++编译器为变量分配内存的策略。

一个 C++程序在内存中所占有的存储空间可分为 3 部分：程序区、静态存储区和动态存储区。

程序区用来存放函数的代码。

静态存储区用来存放静态存储的变量。所谓静态存储是指在程序开始执行时就为变量分配内存，程序执行完毕时才释放这些内存。即在程序执行的整个过程中，静态存储的变量一直占有为其所分配的内存单元，它们的生存期为整个程序的执行期间。

动态存储区用来存放动态存储的变量。所谓动态存储是指在变量的作用域开始处为其分配内存，在其作用域结束处释放这些内存，即动态存储变量的生存期仅在其作用域以内。

在 C++语言中，变量的存储类共有 4 种：自动的、静态的、寄存器的和外部的。

4.4.2.1 自动变量

在说明局部变量时，用关键字 auto 修饰的变量为自动变量。由于 C++编译器默认所有局部变量均为自动变量，所以在实用中基本不使用关键字 auto。例如下面函数 f（）中说明

的变量 i 和 j 都是自动变量。

```
void  f(void)
{
      auto int      i;
      int    j;
      //...
}
```

对于自动变量，为它们分配存储空间及收回它们所占存储空间的工作是由系统自动处理的，这就是自动变量这一叫法的由来。自动变量是动态存储的。

4.4.2.2 静态变量

静态变量是指在说明时用关键字 static 修饰的变量。用 static 修饰的局部变量和全局变量具有不同的含义。

1. 用 static 修饰的局部变量，是要求系统对所说明的变量采用静态存储方式。这一要求通常是出于以下的目的：为了在其作用域之外仍能保留其值，以便下次进入其作用域时能继续使用该值；或者，为能正确地读取返回地址或引用的函数之返回值（这一点将在第 6 章中介绍）。

例 4.8 使用静态局部变量。

```
//      EX4_8.CPP

#include      <iostream.h>
#include      <conio.h>

int     main()
{
      int     count(int);

      do      {
            count(0);
      } while(!kbhit());
      cout << "Function COUTN() have been called " << count(1) << " times.\n";

      return 0;
}

int     count(int i)
{
      static int      c = 0;

      if(i)
            return c;
      else
            ++ c;

      return 0;
}
```

该程序运行时无任何显示，当用户按下任一键时，程序将显示：

Function COUTN() have been called 13350 times

并结束运行。(这里假设当用户按下键时，函数 count（）已经被调用了 13 350 次)。

本程序中，函数 count（）里的变量 c 是用来记录该函数被调用的次数的。若不将它说明为静态的，那么，每当调用该函数时，由于初始化的作用总是将它的值置为 0，从而使得函数 count（）的返回值最大为 1。更重要的是，每当 count（）被调用完毕后，其局部变量 c 就不复存在，当然就起不到记录函数被调用次数的作用了。

应当说明的是，静态变量的初始化仅在变量说明时执行一次，以后，由于该变量已经存在，系统将不再处理该变量的说明语句，当然也就不存在初始化的问题了。另外，变量的静态存储并不改变变量的作用域，例中函数 count（）里的变量 c 在函数之外仍是不可见的。

2. C++语言中全局变量无论是否有存储类修饰均是存储在静态区的。因而用关键字 static 修饰的全局变量并不是对编译系统提出存储该变量之方式的要求，而是限制该变量的可见性仅局限于说明它的文件中。对于由多个文件构成的程序（参见 4.11 节），静态全局变量仅仅在说明它的那个文件中是可访问的，而在其他文件中则是不可访问的。例如：

```
// FILE1.CPP                          // FILE2.CPP
static int    i;                     extern int    i;

int    main()                        void  f(int m)
{                                    {
       i = 5;        // 正确                i = m;          // 错误，i 不可见
       //...                               //...
}                                    }
```

这里，文件 FILE1.CPP 和 FILE2.CPP 共同构成一个程序，前者定义了一个全局变量 i，由于该变量被定义成静态的，所以在后一个文件中尽管使用了 extern 关键字（参见本节“外部变量”），变量 i 仍是不可访问的。

将全局变量说明为静态变量是为了解决在程序的多文件组织中全局变量重名的问题。当一个程序是由多个文件构成时，不同的文件常常是由多人分别编写而成的。这就难免会出现在不同的文件中使用了同名的全局变量。若程序员欲将某一全局变量只限于在自己编写的文件中使用，则应将它说明为静态的。

4.4.2.3 寄存器变量

用关键字 register 修饰的局部变量就是寄存器变量，寄存器变量采用动态存储方式。关键字 register 是指示编译器尽可能地将所说明的变量存放在 CPU 的寄存器中，而不是存放在存储器中。其目的是为了提高对该变量访问的效率。

例 4.9 使用寄存器变量的例子——延时函数。

```
//      EX4_9.CPP

void    delay(int n)
{
        for(register int i = 0; i < n; i ++);
```

}

该函数中 for 循环的循环体为一空语句，即什么都不做，只是拖延一段时间。延时函数在许多场合都有实用价值，根据参数值的不同，该函数所拖延的时间长度也就不同。这里将循环控制变量 i 说明为寄存器的，就是为了提高循环体的运行效率。在实用中，用 register 修饰的变量常常是循环变量。

应说明的是：若在程序中说明的寄存器变量过多，或者在程序运行的某一期间 CPU 暂时没有空闲的寄存器，系统将会把一些寄存器变量当作自动变量处理；反之，若某一期间 CPU 有空闲的寄存器，则编译器的优化功能将会自动地把一些自动变量处理成寄存器变量。因此，在程序设计中很少说明寄存器变量。另外，全局变量与静态变量不得说明为寄存器的，因为这样的说明本身就是矛盾的。

4.4.2.4 外部变量

用关键字 extern 修饰的全局变量为外部变量。外部变量的说明用于具有多文件组织的程序中，它向编译器说明：该变量已在其它文件中定义过了。例如：

```
// FILE1.CPP
int    i;

int    main()
{
       i = 5;
       //...
}
```

```
// FILE2.CPP
extern int    i;

void   f(int m)
{
       i = m;        // 使用 FILE1.CPP 定义的 i
       //...
}
```

不带 extern 说明的全局变量属于定义性说明，因此，若多个文件中有一个文件对某全局变量进行过这样的说明，则在其他文件中不得对该变量进行初始化。因为变量的初始化属于定义性说明。例如：

```
// FILE1.CPP
int    i
//...
```

```
// FILE2.CPP
extern int    i = 5;        // 错误
//...
```

但下例则是正确的:

```
// FILE1.CPP
extern int    i;
//...
```

```
// FILE2.CPP
extern int    i = 5;
//...
```

外部变量说明也可以出现在块作用域中，这时仍属于引用性说明。例如：

```
// FILE3.CPP
void   f(void)
{
       extern int    i;

       cout << i << endl;
}
```

4.5 函数的递归调用

在调用一个函数的过程中又直接或间接地调用该函数本身的这一现象，叫做函数的递归调用。C++语言支持函数的递归调用。

在许多学科中，常常用递归的方法来定义一个概念。例如，在计算机科学中，标识符这一概念用 BNF（Backus-Naur Formula，巴科斯－诺尔范式）方法可表示为

<标识符> ::= <字母> | <标识符><数字> | <标识符><字母>

它的意思是："标识符被定义为：一个字母，或者一个标识符后跟一个数字，或者一个标识符后跟一个字母"。这里用了标识符来定义标识符，这就是一种递归定义。

再如，数学中幂的概念常定义成如下形式：

$$X^n = \begin{cases} 1 & (n=0) \\ X \cdot X^{n-1} & (n>0) \end{cases}$$

而阶乘的概念常被定义成：

$$n! = \begin{cases} 1 & (n=0) \\ n \cdot (n-1)! & (n>0) \end{cases}$$

这些都属于递归定义。而将一个递归的概念提炼成一个适合计算机处理的算法，当然采用递归算法是最为直接的。

例 4.10 递归的阶乘函数。

```
//      EX4_10.CPP

unsigned long       facto(int n)
{
      if(n == 0)
            return 1;                          // A
      return(n * facto(n - 1));                // B
}
```

例 4.11 求 X 整数次幂的递归函数。

```
//      EX4_11.CPP
long  power(int x, int n)
{
      if(n == 0)
            return 1;
      return(x * power(x, n - 1));
}
```

从以上两个例子可以看出，函数的实现部分几乎和概念的定义完全一样。这就使得函数的设计变得非常方便，同时，写出的代码简洁易读，便于理解。

但是，递归函数的执行过程却是相当复杂的。下面以函数调用：

```
nl = facto(3);
```

来说明递归函数的执行过程。

在函数调用 facto（3）（这里暂且称其为对该函数的原始调用）中，由于实参为 3，不等于 0，所以语句 A 不执行而执行语句 B。由于语句 B 中存在有函数调用，其算术表达式的值尚无法计算，所以先将 n 的值（现在是 3）压入堆栈保存起来，然后用值 n - 1（即 2）调用函数 facto（）。这时执行对 facto（）的第一次递归调用。

在对函数 facto（）第一次递归调用时，由于实参为 2，与上述情况相同，于是将值 2 入栈，然后用值 1 调用函数 facto（）。这时执行对 facto（）的第二次递归调用。

在对函数 facto（）第二次递归调用时，由于实参为 1，仍与上述情况相同，于是将值 1 入栈，然后用值 0 调用函数 facto（）。这时执行对 facto（）的第三次递归调用。

在对函数 facto（）进行第三次递归调用时，由于实参为 0，于是执行语句 A，返回值 1。这一值被返回到对 facto（）第二次递归调用时的语句 B 中，由于该语句中算术表达式乘号两边的值都已有结果，可以执行乘法运算，于是就弹出栈顶的数据（这时为 1），与上述返回值相乘后将乘积（为 1）返回，并结束了对 facto（）第二次递归调用。对函数 facto（）第二次递归调用的值被返回到对 facto（）的第一次递归调用中，在这里将该值与这时出栈的数据（值为 2）相乘后，带着乘积（为 2）从对 facto（）的第一次递归调用中返回。这次就返回到对 facto（）原始调用中，函数中的语句 B 用同样的方法计算出 3 与 2 的乘积 6，结束对 facto（）调用并将值 6 返回给主调函数（例中，由变量 n 接受该值）。至此，对 facto（）函数的调用告以结束，而其值 6 即为 3 的阶乘。

从上述递归函数的执行过程可以看出，递归函数执行时其时间开销与空间开销都是非常大的。所以，尽管递归函数具有描述清晰、简洁直观的优点，但由于它的效率极低，因而在实用中常常将递归算法转换成递推方式来编写程序。

将递归函数转换成非递归函数的技术已超出了本书讨论的范围，这里不准备再作进一步地介绍，有兴趣的读者可参阅有关资料。这里需要指出的是，有一些算法若不采用递归方式则极难描述清楚，同时程序设计的难度也相当大。最典型的例子就是 Hanoi 塔问题，该问题可以说只能用递归方法来解决。

Hanoi 塔问题是这样的：相传在天竺国贝拿勒斯的神庙里，安放着一块黄铜板，板上插着 3 根宝石针。创造之神梵天在创造世界时，将 64 个大小不等、中间有孔的金片从大到小地套在第一根针上，形成一个塔形。这就是所谓的 Hanoi 塔，也叫梵塔。不论白天黑夜，都有一个值班的僧侣按照一定的法则将这些金片在 3 根针之间来回移动。法则规定：每次只能移动一个金片，并且无论在哪根针上，都只允许小的金片压在大的上面。梵天预言说，当所有 64 个金片都从第一根针上移到第三根针上时，世界将在一声霹雳中消灭。

例 4.12 Hanoi 塔的求解。

设 3 根宝石针分别为 A，B 和 C，上述问题就可以转化为将 n 个金片从 A 借助 B 移动到 C 上。该问题可以用以下 3 个步骤来解决：

1. 将 A 上 n - 1 个金片借助 C 先移到 B 上；
2. 将 A 上剩下的一个金片移到 C 上；
3. 将 B 上 n - 1 个金片借助 A 移到 C 上。

这实际上是一个递归定义的算法，即用移动 n - 1 来定义移动 n 的解法。根据以上步骤，可以写出以下程序：

```
//      EX4_12.CPP

#include      <iostream.h>

void   Hanoi(int);

int    main()
{
       int     n;

       cout << "Please enter disk number: ";
       cin >> n;
       Hanoi(n);

       return 0;
}

void   Hanoi(int n)
{
       void   MoveTower(int, char, char, char);

       MoveTower(n, 'A', 'B', 'C');
}

void   MoveTower(int n, char from, char used, char to)
// 借助 used 将 n 个金片从 from 移到 to
{
       void   MoveDisk(char, char);

       if(n == 1)
              MoveDisk(from, to);
       else   {
              MoveTower(n - 1, from, to, used);
              MoveDisk(from, to);
              MoveTower(n - 1, used, from, to);
       }
}

void   MoveDisk(char from, char to)
// 将一片金片从 from 移动到 to
{
       cout << from << "-->" << to << endl;
}
```

运行该程序，设输入为 3，则输出为：

```
A-->C
A-->B
```

C-->B
A-->C
B-->A
B-->C
A-->C

输出中，A-->C 表示将 A 针最上面的一片金片移到 C 针上。其余以此类推。

注意：运行该程序时，输入的数不要太大，否则不仅随着数值的增大，运行时间将以几何级数增加，而且有可能引起栈溢出。

这里应强调指出的是：在一个递归程序中，必须存在终止递归的语句，而且应保证该语句在递归调用过程中终究要被执行，以免发生无限递归。

4.6 内联函数

当调用一个函数时，系统将要做许多的工作，主要包括：

1. **保护断点** 将主调函数中的下一条指令的地址入栈；
2. **保护现场** 将一些重要寄存器中的内容入栈；
3. **参数入栈** 如果有参数的话；
4. **执行函数体**；
5. **保存返回值** 如果有返回值的话；
6. **恢复现场** 寄存器内容出栈；
7. **恢复断点** 将主调函数的下一条指令的地址出栈。

这一切都需要一定的时间开销。

在例 4.4 中，给出了一个求两个整数中最大值的函数 max（）。该函数的函数体很小，只有一条语句。在调用该函数时执行函数体所消耗的时间与函数调用时的其他时间开销相比就显得微不足道。如果函数被频繁调用，则附加的时间开销将大得不容忽视。这样，函数调用的时间开销与由于采用了函数而节省的空间开销相比，就显得很不合算。

C++语言为了解决这一矛盾，提供了一种称做“内联函数”的机制。该机制是通过将函数体代码插入到函数调用处，将转子（调用函数）方式改为顺序执行方式来节省程序执行的时间开销，这一过程叫做内联函数的扩展。内联函数实际上是一种用空间换时间的方案。

欲定义一个内联函数，只需在定义函数时，将该函数用关键字 inline 修饰即可。

例 4.13 将例 4.4 中的函数 max（）改写成内联函数。

```
//      EX4_13.CPP

#include      <iostream.h>

inline int      max(int x, int y)
{
      return (x >= y ? x : y);
}

int     main()
```

```
{
    cout << "Please enter two integer: ";

    int     a, b;

    cin >> a >> b;
    cout << "The maximum is " << max(a, b) << endl;              // A

    return 0;
}
```

该例与例 4.4 的功能完全相同，但它们的执行方式却不同。在本例中不再有对 max（）函数的调用，编译器将把该例中的输出语句 A 处理成如下形式：

```
cout << "The maximum is " << (a >= b ? a :b) << endl;
```

请读者自己比较一下两者的区别。

使用内联函数时应注意：

1. 在 C++程序中，除了在函数体中含有循环、switch 分支和复杂嵌套的 if 语句的函数外，所有的函数均可以被说明为内联的。

2. 内联函数的定义必须出现在对该函数的调用之前，如例 4.13 所示。这是因为编译器在对函数调用语句进行代换时，必须事先知道代换该语句的代码是什么之故。若像例 4.4 那样，就是在函数原型和函数定义处均加上关键字 inline 都不行。

3. 由于计算机的资源总是有限的，使用内联函数虽然节省了程序运行的时间开销，但却增大了代码占用内存的空间开销。因此在具体编程时应仔细地权衡时间开销与空间开销之间的矛盾，以确定是否采用内联函数。

4. 与处理 register 变量相似，是否对一个内联函数进行扩展完全由编译器自行决定。因此，说明一个内联函数只是请求而不是命令编译器对它进行扩展。

4.7 带有缺省参数的函数

C++语言允许在函数说明或函数定义中为形参预赋一个缺省的值，这样的函数就叫做带有缺省参数的函数。在调用带有缺省参数的函数时，若为相应形参指定了实参，则形参将使用实参的值；否则，形参将使用其缺省值。这就大大地方便了函数的使用。

例 4.14 将例 4.9 中的延时函数改为带有缺省参数的函数。

```
//      EX4_14.CPP

#include      <iostream.h>

void    delay(int n = 100);

int     main()
{
    cout << "Delaying 500 units...\n";
    delay(500);                                                 // A
```

```
    cout << "Now, delaying 100 units...\n";
    delay();                                          // B
    cout << "That's all for this program.\n";

    return 0;
}

void  delay(int n)
{
    for(; n > 0; -- n);
}
```

函数 delay（）是一个带有缺省参数的函数，其形参 n 给出了延时的长度。在函数原型中为形参 n 设定缺省值为 100。程序中行 A 用实参 500 来调用该函数，函数就使用该值延时 500 个单位；而行 B 在调用函数时没有给出实参，则函数就使用其缺省值延时 100 个单位。

使用带有缺省参数的函数时应注意：

1. 缺省参数的说明必须出现在函数调用之前。这就是说，如果存在函数原型，则形参的缺省值应在函数原型中指定；否则在函数定义中指定。另外，若函数原型中已给出了形参的缺省值，则在函数定义中不得重复指定，即使所指定的缺省值完全相同也不行。

2. 对形参缺省值的指定可以是任意的初始化表达式，甚至可以包含函数调用。如下例所示。

例 4.15 缺省参数为函数调用的例子。

```
//    EX4_15.CPP

#include    <iostream.h>

int   max(int, int);
void  delay(int n = max(100, 100))
{
    for(; n > 0; n --);
}

int   main()
{
    int   a, b;

    cin >> a >> b;
    cout << "Delaying the bigger between a and b...\n";
    delay(max(a, b));
    cout << "Now, delayint 500 units...\n";
    delay(500);
    cout << "And now, delaying 100 units...\n";
    delay();
    cout << "That's all for this program.\n";

    return 0;
}
```

```
int     max(int x, int y)
{
        return(x >= y ? x : y);
}
```

该程序的执行过程请读者自行分析。

3. 若函数具有多个形参，则缺省参数必须自右向左连续地定义，这是由于 C++语言在函数调用时参数是自右至左入栈这一约定所决定的。例如：

```
int     f(int a, float b = 5.0, char c = '.', int d = 10);
int     f(int, float = 5.0, char = '.', int = 10);          // 仅函数原型
```

而以下的说明则是错误的:

```
int     f(int a = 10, float b = 5.0, char c = '.', int d);
int     f(int a, float b = 5.0, char c, int d = 10);
```

4. 如果几个函数说明出现在不同的作用域内，则允许分别为它们提供不同的缺省值。例如：

```
void  f(int, int = 0);                          // 全局说明

void  f1(void)
{
      void  f(int = 5, int = 5);                // 局部说明

      f();                                      // 按局部说明调用函数 f()，等价于 f(5, 5);
}

void  f2(void)
{
      f(3);                                     // 按全局说明调用函数 f()，等价于 f(3, 0);
}

void  f(int a, int b)
{
      // 函数 f()的定义
}
```

4.8 参数数目可变的函数

至今为止所介绍的函数其参数个数都被明确地指定了，在调用一个函数时，实参的个数必须与形参的个数完全相等。就是在调用一个带有缺省参数的函数时，这种参数相等的事实仍是存在的。然而在有些应用中，人们希望一个函数的参数个数是可变的，以满足不同场合的需要。C++语言为满足这种需要提供了可能，这就是允许定义参数数目可变的函数。

先看一个例子。

例 4.16 参数数目可变的函数。

```
//      EX4_16.CPP

#include      <iostream.h>
#include      <stdarg.h>

int     max(int, int...);

int     main()
{
        int     a, b, c, d, e;

        cout << "Enter five integers, separate with space: ";
        cin >> a >> b >> c >> d >> e;
        cout << "The bigger between a and b is " << max(2, a, b) << endl;
        cout << "The maximum in the five integers is ";
        cout << max(5, a, b, c, d, e) << endl;

        return 0;
}

int     max(int num, int integer...)
{
        va_list ap;                                     // A
        int     m = integer;

        va_start(ap, integer);                          // B
        for(int i = 1; i < num; i ++)   {
                int     t = va_arg(ap, int);            // C

                if(t > m)
                        m = t;
        }
        va_end(ap);                                     // D

        return m;
}
```

例 4.16 中的函数 max（）就是一个参数数目可变的函数，其参数表中的省略号“...”表示它的后边可能还带有一些参数。

在编写一个参数数目可变的函数时，必然要涉及到系统头文件 stdarg.h 中所说明的一个类型 va_list 以及 3 个宏（宏将在 4.10.2 中介绍）va_start（），va_arg（）和 va_end（）。va_list 与 int，char 等相似，是一个 C++系统预定义的一个数据类型，它仅用于参数数目可变的函数中来说明一个具有这种数据类型的变量，以便读取参数表中的各个实参的值。程序中行 A 所说明的变量 ap 就是这样的变量。

宏 va_start（）具有以下的形式：

```
void  va_start(va_list argPtr, last parm);
```

该宏带有两个参数：第一个参数是一个 va_list 型的变量；第二个参数应为参数数目可变的函数之形参表中最后一个固定参数的名字。该宏用来初始化其第一个实参（本例中为变量 ap），使它作好读取其第二个参数的准备。程序中行 B 就是使变量 ap 作好读取 integer 参数值的准备。

宏 va_arg（）具有以下的形式：

```
type   va_arg(va_list argPtr, type);
```

该宏也带有两个参数，其第一个参数也是一个 va_list 型的变量，第二个参数应当是一个已存在的数据类型（包括基本数据类型和导出数据类型）。该宏的作用是读取第一个参数已准备好的数据，并以第二个参数所代表的数据类型返回该数据的值，同时使第一个参数指向实参表中的下一个参数，为下一次读数作好准备。如程序中行 C 就是将变量 ap 所指的实参值以 int 型返回并赋给局部变量 t，同时使 ap 指向实参表中的下一个实参。

宏 va_end（）具有以下的形式：

```
void   va_end(va_list argPtr);
```

该宏只有一个 va_list 型的参数，它的作用是清除其参数所代表的变量，以使参数数目可变的函数可以正常返回。如程序中行 D 就是用该函数来清除变量 ap。注意：若在一个参数数目可变的函数中忘记使用或者不正确地使用该宏，都有可能引起系统的崩溃（典型现象就是死机）。

应当说明的是，在一个参数数目可变的函数中，必须有一个能够具体确定实参个数的方法。在例 4.16 中，函数 max（）的第一个参数就是用来直接给出参数的个数的。当然也可以利用其他方法，第 6 章中将给出一个这样的例子。

4.9 函数重载

在 C++语言库函数中，有以下 4 个功能相似的函数：

```
int     abs(int);
double      fabs(double);
long   labs(long);
double      cabs(struct complex);
```

这些函数的函数原型均在头文件 math.h 中说明，它们的功能依次为求整型量、实型量、长整型量和复数的绝对值。

这些函数虽然方便了程序员编写程序，但同时又增加了程序员的记忆难度。在日常数学中，我们并不说“求某实数的绝对值，求某整数的绝对值”，而只说“求某数的绝对值”。即把求绝对值的方法看成是一个通用的方法。事实上 C++语言提供了满足这一要求的手段，它就是 C++语言的重载（Function Overloading）功能。

在 C++语言中，重载包括函数重载和运算符重载，本节介绍函数重载，运算符重载将在第 12 章中介绍。

例 4.17 函数重载的例子。

```
//      EX4_17.CPP

#include      <iostream.h>

int     abs(int);
double        abs(double);
long    abs(long);

int     main()
{
        int     i;
        double          d;
        long    l;

        cout << "Enter an integer, a real and a long integer successively:\n";
        cin >> i >> d >> l;
        cout << "Their absolute value are: "
                << abs(i) << '\t'                                   // A
                << abs(d) << '\t'                                   // B
                << abs(l) << endl;                                  // C

         return 0;
}

int     abs(int num)                                                // A'
{
        return(num > 0 ? num : -num);
}

double        abs(double num)                                       // B'
{
        return(num > 0 ? num : -num);
}

long    abs(long num)                                               // C'
{
        return(num > 0 ? num : -num);
}
```

本例中，定义了 3 个同名的函数 abs（），它们分别为求整型量、实型量和长整型量绝对值的函数。在 main（）函数中，行 A，B 和 C 分别调用了函数 A'，B'和 C'。

重载了的函数既然名子相同，那么编译器是根据什么确定一次函数调用是调用的哪一个函数呢？编译器是根据函数的参数来确定应该调用哪一个函数的。当实参为 int 型时，它就调用函数 A'，当实参为 float 或 double 型时，调用函数 B'，而当实参为 long int 时，就调用函数 C'。因此，重载了的函数之间必须在参数的类型或个数方面有所不同，否则有可能导致一个编译错误。比如，以下的函数说明均属于合法的重载：

```
int     func(int);
int     func(int, int);
void    func(int, char);
```

```
void   func(char, int);
```

而下面的说明将导致一个编译错误：

```
int    func(int);
void   func(int);
```

这是由于 C++语言允许忽略函数的值或用与函数返回值类型不同的变量来接收函数的值，若几个函数说明仅仅是返回值不同，编译器就无法确定调用的是哪一个重载了的函数。

以下的说明：

```
int    func(int);
int    func(int);
```

虽是合法的，但不是重载。它们仅仅是对同一个函数的多次引用性说明。

重载函数从一个方面体现了 C++语言对多态性的支持，即实现了 OOP 技术中所谓“一个名字，多个入口”，或称“同一接口，多种方法”的多态性机制。

4.10 编译预处理

C++语言提供了编译预处理功能，使得编译器在对源程序运行正式编译前，可以根据预处理指令先做一些特殊的处理工作，然后将预处理的结果与源程序一起进行编译。C++语言的预处理功能包括 3 个方面：嵌入指令、宏定义和条件编译指令。

编译预处理指令不属于 C++语言的语法范畴，因此，为了和 C++语句区别开来，预处理指令一律以符号“#”打头，以“回车”结束，即每条预处理指令必须单独占用一行。若一行书写不下，则在键入回车符之前先键入一个续行符“\”（注意：在预处理指令中该符号不再是转义字符）。

4.10.1 嵌入指令

嵌入指令的一般形式为

#include <file_name> 或 ***#include "file_name"***

其中：file_name 是一个文件名。该文件名可以带有路径，例如：

```
#include    <myfile.h>
#include    "c:\mydir\prog.h"
```

嵌入指令是用来指示编译器将指定的源文件插入到该指令所在的位置处后再进行编译。例如：

```
// MYMAIN.CPP

#include    "myprog.cpp"

int    main ()
```

```
// MYPROG.CPP

void  f(void)
{
      cout << "Function is called \ n";
```

```
{                                                    }
        f();

        return 0;
}
```

这样，当编译器在编译 MYMAIN.CPP 时，会把它变成如下形式：

```
// MYMAIN.CPP

// MYPROG.CPP

void  f()
{
        cout << "Functio is called \ n";
}

int     main()
{
        f();

        return 0;
}
```

然后才进行编译的。当然，这一过程是由编译器临时进行的。由于嵌入指令并不真正改变用户的源程序，所以，一般情况下这一变化用户是看不到的。

应当指出的是：上例仅仅是为了说明利用嵌入预处理指令可以将任何一个文本文件嵌入到另一个文本文件中，然而，在实用中通常只在源程序文件（.CPP）中嵌入头文件（.H）。实际上，在一个.CPP 文件中嵌入另一个.CPP 文件尽管在理论上是没有问题的，但若组织的不当常常会导致编译失败。

在嵌入指令中，尖括号与双引号所起的作用是不同的。使用尖括号是指示编译器在系统的包含文件目录中去寻找所指定的文件，通常该目录为 INCLUDE 子目录。这一用法常用来嵌入 C++系统的头文件，如 iostream.h，stdarg.h 等；使用双引号是指示编译器首先在当前工作目录下寻找所指定的文件，如果没有找到，再到系统的包含文件目录中去寻找。这一用法常用来嵌入程序员自定义的头文件。

无论采用哪种指示形式，如果编译器按上述搜索方式没能找到指定文件，都会给出一个错误信息并终止编译。因此，这里建议读者：若欲嵌入系统头文件，就使用尖括号；若欲嵌入程序员自己编写在头文件，就使用双引号并给出完整的路径。

4.10.2 宏

预处理指令#define 用来定义一个宏。宏提供了一种词法符号替换机制，它们可带也可不带类似函数参数的形式参数。

4.10.2.1 无参宏定义

定义一个无参宏的一般形式为

#define macro_name <text>

其中：macro_name 应是一个合法的 C++标识符，它是程序员所定义的宏的名字，简称宏名或宏；而可选的 text 则是一段正文，叫做替换正文，它就是程序员对宏名的实际定义。例如，由于 C++语言中没有逻辑型变量，在实用中常常做如下的宏定义：

```
#define    FALSE    0
#define    TRUE     1
```

它们的意思是：将标识符 FALSE 定义为 0，将 TRUE 定义为 1。

有了以上的宏定义，编译器在进行编译时，首先将源程序中所有的 FALSE 换成 0，TRUE 换成 1，然后才进行正式编译。这一过程叫做宏替换，例如：

```
if(result == TRUE)                    // A
```

将被替换成：

```
if(result == 1)                       // B
```

当然，这一替换是在系统内部进行的，不会改变源程序的内容。

再如，宏定义：

```
#define    MSG "Please enter an integer: "
```

将指示编译器把语句：

```
cout << MSG;
```

替换成：

```
cout << "Please enter an integer: ";
```

使用宏定义常常是出于两方面的考虑：其一是为了提高程序的可读性，比如，上述的行 A 就比行 B 容易使人看的明白些；其二是为了方便源程序的修改，比如，在有些应用中希望以 - 1 代表假，我们不必查遍整个程序将所有的 0 都改成 - 1（其实，也许源程序中有些 0 本身就不是代表假的，这样的改动还可能引起程序的逻辑错误），只要有了上述的宏定义，我们只须将上述宏定义中的 0 改成 - 1，其他的工作就让编译器来做了。

在习惯上，人们常将宏名写成大写形式，以示与其他标识符的区别。但这不是规定，C++语言只要求宏名的拼写方式满足对标识符的要求即可。

对于宏定义，这里需要说明几点：

1. 宏定义必须以“回车”结束，编译器会将宏定义中从正文开始处到回车符前的所有内容来替换宏名的。例如：

```
#define    TRUE     1;
```

将使得编译器把源程序中所有的“TRUE”替换成“1;”。

2. 宏定义中的 text 部分可以为空，比如：

```
#define    MYHEAD_H
```

在实用中，常常用这种方法建立了一个标志，以供条件编译指令来检测。

3. text 部分也可以是 C++的关键字，比如：

```
#define     U_L  unsigned long int
```

这样，在程序中就可以用以下的形式来定义无符号长整型变量，以简化程序的书写：

```
U_L  l;
```

4. 一个宏名可以出现在另一个宏的替换正文中，例如：

```
#define     ONE       1
#define     TWO       ONE + ONE
#define     THREE     ONE + TWO
```

当编译器对这些宏进行替换时，替换过程是逐级进行的，比如：

```
i = THREE;
```

将被按以下顺序逐步替换

```
i = ONE + TWO;  →  i = 1 + ONE + ONE;  →  i = 1 + 1 + 1;
```

但是，在这种情况下，对以下语句的替换可能违背了程序员的原意：

```
i = THREE * 3;
```

程序员的意思可能是向变量 i 赋以值 9，但由于编译器对该语句替换的结果为

```
i = 1 + 1 + 1 * 3;
```

最终导致赋给 i 的值为 5。为了防止出现这种问题，可采用以下的写法：

```
i = (THREE) * 5;
```

或干脆改变宏定义：

```
#define     TWO       (ONE + ONE)
#define     THREE     (ONE + TWO)
```

后一种方法效果更好。

5. 编译器不对替换正文中的语法错误进行检查，而是留到替换后对源文件进行编译时检查。例如：

```
#define     A_MSG     "This is an error" i * i

cout << A_MSG;                                    // A
```

编译器将指出语句 A 有语法错误。

6. 宏名虽是一个标识符，但它不是变量，因此不能当作一个变量使用。事实上，宏名更象一个常量，比如，宏 ONE 实际上就是 1；而宏 MSG 实际上就是"Please enter an integer"。因此，宏也被称作符号常量。

7. 定义一个作为字符串常量的宏时应特别注意它的定义和用法。例如：

```
#define     MSG This is a message
cout << "MSG";
```

输出则为

```
MSG
```

这是因为双引号是 C++语言的字符串常量定界符，编译器将双引号中的内容“MSG”当作一个字符串常量而不是当作宏来处理的。而

```
cout << MSG;
```

则导致一个编译错误，这是因为该语句经宏展开后变成了

```
cout << This is a message;
```

但插入运算符右边既不是一个变量，又不是一个常量，也不是一个表达式。

4.10.2.2 定义带参的宏

带参宏定义的一般形式为

#define macro_name(arg_list) text

其中：arg_list 为宏定义的参数表，在这里可以给出宏定义的形参。例如

```
#define     MAX(x, y)        x >= y ? x : y
```

就定义了一个带有两个形参 x 和 y 的宏 MAX。在程序中使用该宏时，就像调用一个函数：

```
cout << MAX(i, j);
```

该语句将被编译器替换成

```
cout << i >= j ? i : j;
```

但这将导致一个语法错误，其原因请读者自行分析。特别是当实参为一表达式时，则在许多地方都可能造成语法错误或逻辑错误。为了防止这种问题出现，应将原定义改写成

```
#define     MAX(x, y)        ((x) > (y) ? (x) : (y))
```

带参宏的定义往往比较长，有时需要书写成多行。但由于“回车”符意味着一个宏定义的结束，所以在需要换行时，应加一个续行符“\”。比如：

```
#define     MAX(x, y)        ((x) >= (y) ?\
                             (x) : (y))
```

总之，宏定义存在有许多不安全因素，在实用中最好换用 const 变量和内联函数。例如：

```
const int    FALSE = 0;
const int    TRUE = FALSE + 1;
inlien int   MAS(x, y)
{
    return (x >= y ? x : y);
```

}

4.10.2.3 #undef 指令

#undef 指令的一般形式为

#undef macro_name

其作用是使对宏 macro_name 的定义无效，以供条件编译指令用来测试某宏名当前是否被定义，或者准备重新定义该宏。例如：

```
#define     BLOCK_SIZE    512
// 对宏 BLOCK_SIZE 的某些使用
#undef      BLOCK_SIZE
#define     BLOCK_SIZE    128
```

宏 BLOCK_SIZE 原被定义为 512，当需要重新将它定义为 128 时，必须先用#undef 指令使其无效，然后才可重新定义。当 BLOCK_SIZE 无效时，对该标识符的访问将是非法的。

4.10.3 条件编译指令

C++语言提供条件编译指令，以控制编译器有选择地对源程序运行编译。条件编译指令指示编译器可以忽略某些以符号“#”打头的行（除了#if，#ifdef，#ifndef，#elif 和#endif 指令），以及由于条件编译指令的结果而不用参加编译的行。

4.10.3.1 #if，#elif，#else 和#endif 条件指令

#if，#elif，#else 和#endif 条件指令的工作过程与 C++的 if 语句相似。其中，#elif 相当于 else if，而#endif 则用来结束条件序列。

例 4.18 一个条件编译的例子。

```
//      EX4_18.CPP

#include      <iostream.h>

#define       NUM  100

int     main()
{
        #if     NUM >= 100
                cout << "This line is compiled.\n";              // A
        #else
                cout << "This line is not compiled.\n";          // B
        #endif

        return 0;
}
```

该程序的输出为

```
This line is compiled.
```

这说明编译器根据给定条件编译了行 A 而忽略了行 B。实际上编译器将该程序处理成了如下的形式：

```
#include    <iostream.h>

int    main()
{
       cout << "This line is compiled. \ n";

       return 0;
}
```

再看一个复杂些的例子。

例 4.19 根据给定条件输出相应国家货币名。

```
//      EX4_19.CPP

#include      <iostream.h>

#define       US          0
#define       ENGLAND     1
#define       CHINA       2
#define       COUNTRY     CHINA                    // A

int     main()
{
       #if    COUNTRY == US
              cout << "Dollar\n";
       #elif  COUNTRY == ENGLAND
              cout << "Pound\n";
       #elif  COUNTRY == CHINA
              cout << "Yuan\n";
       #else
              cout << "Error!\n";                  // B
       #endif

       return 0;
}
```

该程序输出为

```
Yuan
```

4.10.3.2 #ifdef，#ifndef 条件指令和 defined 运算符

#ifdef 和#ifndef 条件指令用来测试某个宏是否已经定义，从而确定编译器的下一步操作。例如：

```
// AHEAD.H

#ifndef     _AHEAD_H_
#define     _AHEAD_H_
```

```
// 文件 AHEAD.H 中的具体内容

#endif
```

这种形式常出现在一个头文件中，以保证在程序的多文件组织（参见下节）中不至于出现在多个文件中对同一个名子的重复说明。

应当说明的是：在 C++语言中保留#ifdef 和#ifndef 条件指令只是为了保持与 C 语言的兼容，它们今后可能被淘汰掉。因此，C++语言鼓励程序员用运算符 defined 来进行标识符的测试。在 C++中，#ifdef 的等价表示为#if defined 或简写为#defined；而#ifndef 的等价表示为#if !defined。因此，上例最好改写为：

```
// AHEAD.H

#if !defined _AHEAD_H_
#define     _AHEAD_H_

// 文件 AHEAD.H 中的具体内容

#endif
```

4.10.3.3 #error 指令

#error 指令通常放在预处理指令的条件中，以便捕捉一些不可预料的编译条件。在正常情况下，条件是假的。若条件为真，编译器将输出一条错误信息并终止编译。该指令的一般形式为

```
#error      errmsg
```

其中：errmsg 为程序员欲使编译器发出的报错信息。如果该指令执行，则将在显示器上显示以下形式的信息：

```
error: filename line#: Error directive: errmsg
```

其中：filename 为正在编译之文件的文件名；而 line#为出错行的行号。例如，若将例 4.19 中的行 A 改为

```
#define     COUNTRY JAPAN
```

并将行 B 改为

```
#error Country name is wrong!
```

则编译器将会在屏幕上显示

```
error: EX4_19.CPP line 19: Error directive: Country name is wrong!
```

并终止编译。

4.11 程序的多文件组织

一个具有一定规模的程序常常是由多人合作编写而成的。即使全部由一个人完成，也常

常为了方便设计、便于调试，而将一个大的程序分写成多个程序文件。每个程序文件就叫做一个文件模块，简称模块。这就是程序的多文件组织。

4.11.1 连接属性

当一个程序是由多个模块组成时，一个模块中的全局名字（变量名、函数名、对象名等）的连接属性决定了该名字是否可以被其他模块所访问。连接属性有两种：内部连接和外部连接。

4.11.1.1 内部连接

内部连接也称静态连接。具有内部连接的名字局部于它所在的模块，只能被同一模块中的函数所访问。所有静态全局名字都具有内部连接属性。例如：

```
// FILE1.CPP
static int     sum(int x, int y)
{
       return(x + y);
}
```

```
// FILE2.CPP
int    main()
{
       int    sum(int, int);

       int    i = sum(5, 9);              // 错误

       return 0;
}
```

这里，设整个程序由文件 FILE1.CPP 和 FILE2.CPP 共同组成。函数 sum（）在文件 FILE1.CPP 中定义，并说明成静态的。在文件 FILE2.CPP 中，尽管给出了 sum（）的函数原型，但它仍旧不能调用该函数。除非在 FILE2.CPP 中给出该函数的定义。

4.11.1.2 外部连接

具有外部连接属性的名字在其它模块中经引用性说明后就可以被访问。所有外部全局名字都具有外部连接属性。比如，将上例 FILE1.CPP 中的关键字 static 去掉或换成 extern，同时将 FILE2.CPP 中的函数说明前冠以 extern，这样，文件 FILE2.CPP 中的函数调用就是合法的了。

通常，为了使用上的方便，常常将一些具有外部连接属性的名字写在一个称为头文件的源文件中，而在使用这些名字的模块中利用嵌入指令包含该头文件。头文件可以认为是源程序文件的一个辅助文件，程序员常常将程序中用到的一些函数的原型、宏、全局变量以及程序员定义的数据结构集中地写在一个文件中，供所有用到这些程序要素的源程序文件共同使用。

例 4.20 使用头文件。

```
//      EX4_20.H

#if !defined   _EX4_20_H_
#define        _EX4_20_H_

extern int     sum;                        // 全局变量
```

```
int     add(int, int);                          // 全局函数
void    output(void);

#endif


//      EX4_20.CPP

#include      "ex4_20.h"

int     sum;                                    // A

int     main()
{
        sum = add(5, 9);
        output();

        return 0;
}


//      EX4_20A.CPP

#include      "ex4_20.h"

int     add(int x, int y)
{
        return (x + y);
}


//      EX4_20B.CPP

#include      "ex4_20.h"
#include      <iostream.h>

void    output(void)
{
        cout << sum << endl;
}
```

该程序由 EX4_20.H，EX4_20.CPP，EX4_20A.CPP 和 EX4_20B.CPP 4 个独立的文件组成，各模块中所共用的名字 sum，add（）和 output（）在头文件 EX4_20.H 中说明。这里，虽然全局名子 add 和 output 没有用关键字 extern 修饰，但由于它们被嵌入到各模块中，从而使得它们事实上具有了外部连接属性。

在本例的头文件 EX4_20.H 中使用了条件编译指令，这是很有必要的，其目的是为了阻止对该文件的重复嵌入。比如，若程序员欲将上述 3 个.CPP 文件进行整体编译而把文件 EX4_20.CPP 改写为：

```
// EX4_20.CPP

#include    "ex4_20.h"
#include    "ex4_20a.cpp"
#include    "ex4_20b.cpp"

int    sum;

int    main()
{
       sum = add(5, 9);
       output();

       return 0;
}
```

这样，就可能将文件 EX4_20.H 嵌入 3 次。由于在头文件 EX4_20.H 中使用了条件编译预处理指令，当第一次嵌入该文件时，就使得宏_EX4_20_H_有了定义，从而导致文件 EX4_20A.CPP 和 EX4_20B.CPP 中的嵌入指令什么都不做。另外，文件 EX4_20.H 中将全局变量 sum 说明成外部的也是非常必要的。因为无 extern 修饰的全局变量属于定义性说明，若去掉该关键字，当采用分割编译时，将会因该变量定义性说明的重复而导致一个连接错误。然而，在 C++语言中，任何一个名字都必须有一个定义性说明，文件 EX4_20.CPP 中的行 A 就是对全局变量 sum 的定义性说明。

4.11.2 分割编译

当一个程序是由多个模块组成的时，可以按一个个源文件为单位将它们分别编译成目标文件（除了头文件），然后再用连接器将将这些目标文件连接成一个可执行文件。这样的编译过程称为分割编译。

分割编译的好处在于：若一个模块的内容需要修改时，只需对该模块重新编译，而不必重新编译全部源程序，从而提高了程序的生成速度，并为软件的维护和扩充提供了便利。另外，分割编译也为各模块的单独调试提供了方便，从而便于保证各模块的可靠性。

以例 4.20 为例，程序员可以先编辑 EX4_20.H 文件并将其存盘；然后分别编辑其它 3 个.CPP 文件并对它们进行编译。在对它们进行编译的过程中，起码可以发现并改正源程序中的语法错误。每个源程序文件在编译无误的情况下可以将它们存盘。最后，只须将编译器生成的 3 个.OBJ 文件连接即可。

习题

4-1 编译、连接并运行本章中所有例题。

4-2 将第 2 章中习题 2-5～2-7 改写成函数，注意它们分别应该带哪些参数。

4-3 编写一个主（main）函数，分别用以下 3 种组织方式调用例 4.1 中的阶乘函数：

1. 将阶乘函数的定义放在主函数的定义之前；

2. 将阶乘函数的定义放在主函数的定义之后；

3. 将阶乘函数的定义和主函数的定义分别放在两个独立的源程序文件中，将阶乘函数的原型放在一个头文件中。

4-4 将第 3 章中的习题 3-5～3-7 改写成函数，并编写主函数调用它们。

4-5 将第 3 章中的习题 3-8～3-9 改写成函数，使它们能够分别输出指定数以内的水仙花数和 Fibonnaci 数列。

4-6 编写二次方程求根函数。

4-7 编写两个函数，分别求两个整数的最大公约数和最小公倍数。

4-8 编写一个递归函数，输出 Fibonnaci 数列的前 n 项。

4-9 当 x > 1 时，Hermite 多项式的定义为：

$$H_0(x) = 1$$
$$H_1(x) = 2x$$
$$H_n(x) = 2x \cdot H_{n-1}(x) - 2(n-1) \cdot H_{n-2}(x)$$

试编写输出 Hermite 多项式对应变量 x 的前 n 项值的递归函数和非递归函数。

4-10 将例 4.17 中的 3 个重载函数改写成内联的，

4-11 将第 2 章习题 2-7 改写成带有缺省参数的函数，其缺省参数为 0。

4-12 编写一个参数数目可变的函数，计算 n 个整数之和。

4-13 定义一个带有 3 个参数的宏 MIN，该宏的值为 3 个参数中最小者的值。考虑如何设计该宏使其在任何情况下都不出现二义性。

第5章 数组

C++语言已预定义了一些基本数据类型，但在实用中，仅用这些基本数据类型并不能满足实际需要。为此，C++又提供了导出数据类型。导出数据类型是由基本数据类型构造而成的。C++语言中的导出数据类型有：数组、结构、联合和类。本章介绍数组，其他导出数据类型将在后继章节中介绍。

5.1 数组

数组是对现实世界中具有相同性质的有序事物集合的抽象，比如数轴上有限点的集合、三维空间中有限点的集合等。前者可以用一维数组表示，而后者可用多维数组表示。

C++语言中的数组（也叫数组变量）是一组相同类型的变量之集合，其中的每个变量称为该数组的元素变量，简称元素。数组中每个元素都有与其对应的下标以标明该元素在数组中的位置，对数组元素的访问通常也是借助下标来进行的，因此，元素也称为下标变量。

5.1.1 一维数组

具有一个下标的数组叫做一维数组。

5.1.1.1 一维数组的说明

说明一维数组的一般形式为

<storage> type array_name[constant];

其中：type 是除 void 类型以外的任一基本数据类型，它指出数组中所有元素的数据类型；array_name 是用户为所说明的数组变量起的变量名；而 constant 为一非负的整型常量，它指出了所说明的数组的大小，即该数组所能容纳元素的个数。例如：

```
int   iarr[20];
float farr[5];
char  carr[80];
```

就分别说明了一个包含有 20 个元素的整型数组 iarr，一个包含 5 个元素的实型数组 farr 和一个包含 80 元素的字符型数组 carr。

5.1.1.2 访问一维数组的元素

像普通变量一样，一个数组一经定义，就可以在程序中访问它的元素。对数组元素的访问一般是通过下标运算符“[]”和下标值来进行的。例如：

iarr[5]

就是访问整型数组 iarr 中，下标值为 5 的那个元素。

应当说明的是：在 C++语言中，数组下标总是从 0 开始的，所以上述访问实际上访问的是 iarr 的第 6 个元素。另外， C++系统对数组的下标超界不做任何检查，下标超界检查应由程序员自己负责。比如：

i = iarr[20];

实际上是欲将 iarr 中根本不存在的第 21 个元素赋给变量 i，对此，编译器并不给出任何错误信息，但变量 i 得到的却是一个错误的值。

例 5.1 用一维数组存放某厂的月产值、年产值和年均产值。

```
//      EX5_1.CPP

#include      <iostream.h>

int     main()
{
        float   output_value[14], sum = 0;

        cout << "Enter month output value sequentially:\n";
        for(int i = 0; i < 12; i ++)     {
                cin >> output_value[i];                                  // 输入各月的产值
                sum += output_value[i];                                  // 求各月产值的和
        }
        output_value[12] = sum;                                          // 存储年总产值
        output_value[13] = sum / 12;                                     // 计算并存储年均产值
        cout << "The year output value is " << output_value[12] << endl;
        cout << "The average output value is " << output_value[13] << endl;

        return 0;
}
```

在例 5.1 中，说明了一个包含 14 个元素的实型数组 output_value，该数组的前 12 个元素用来存放一年中 12 个月的月产值，第 13 个元素存放一年的总产值，而最后一个元素存放年均产值。若没有数组这一导出数据类型，就本例而言至少需要说明 3 个实型变量，如果还要保存各月产值以供其它地方使用的话，则需要说明 14 个实型变量。

5.1.1.3 一维数组的初始化

与普通变量一样，数组也可以在说明的同时为各个元素指定初值。但与普通变量初始化不同的是数组的初始化数据必须用一对花括号括起来。比如：

int   ia[5] = {1, 2, 3, 4, 5};

就将 ia[0]～ia[4]分别初始化为 1～5。

在对数组进行初始化时，允许对数组中的个别元素不赋初值。数值型数组未初始化的元素由系统自动初始化为 0。例如：

```
int    ia[5] = {1, 2};
```

将使得 ia[0]～ia[4]的值分别 1，2，0，0，0。而

```
int    ia[5] = {, , 3, 4};
```

则使得 ia[0]～ia[4]的值分别 0，0，3，4，0。

字符型数组中未初始化的元素则由系统自动初始化为空字符。空字符即 ASCII 码表中值为 0 的字符。比如：

```
char  ca[5] = {, , 'c', 'd',};
```

就说明了一个包含有 5 个元素的字符型数组，其第 1，2，5 元素为空字符，而第 3，4 元素被初始化为字符 'c' 和 'd'。

若对数组中所有元素都给出了初值，则可省略数组说明中的常量 constant。例如：

```
int    ia[] = {1, 2, 3, 4, 5};
```

就说明了一个包含有 5 个元素的数组 ia，它各元素的值分别为 1～5。

5.1.1.4 一维数组的存储形式

数组在内存中是连续存放的。每个元素占据着其类型规定长度的连续内存单元，其后紧跟着它的后继元素。比如，char ca[5]，int ia[3]和 float fa[2]在内存中的存储形式就如图 5-1 所示。

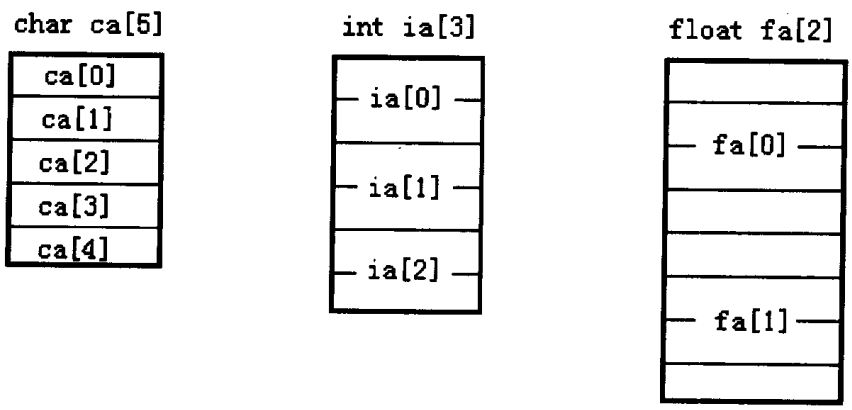

图 5-1　一维数组的存储形式

5.1.2 多维数组

具有多个下标的数组叫做多维数组。多维数组可以看作是一维数组的组合，即元素为数组的数组。

5.1.2.1 多维数组的说明与访问

说明多维数组的一般形式为

<storage> type　　array_name[constant$_1$][constant$_2$]...[constant$_n$];

其中：方括号的对数称作数组的维数，自左向右地分别称作第 1 维、第 2 维、..... 、第 n 维。在实用中，极少用到三维以上的多维数组。例如：

```
int    iarr[3][4];
```

就说明了一个名为 iarr 的二维整型数组，它的第 1 维包含 3 个元素；第 2 维包含 4 个元素。也可以这样来看待 iarr：它是一个包含有 3 个元素的数组，其每个元素又是一个包含有 4 个元素的数组。该数组中共包含了 3×4 即 12 个元素。再如：

```
float  farr[5][3][2];
```

则说明了一个三维元素个数分别为 5，3，2 的三维实型数组 farr。它共包含了 30 个元素。

访问一个多维数组中的元素时，必须给出其每一维的下标，以将访问落实到一个具体的元素上。例如：

```
i = iarr[2][2];
```

表示将 iarr 的第 9 个元素的值赋给变量 i。

例 5.2 建立一个九九表。

```
//      EX5_2.CPP

#include      <iostream.h>
#include      <iomanip.h>

int     main()
{
        int     tab[9][9], i, j;

        for(i = 0; i < 9; i ++)  {
                for(j = 0; j < 9; j ++)
                        tab[i][j] = (i + 1) * (j + 1);
                cout << setw(4) << (i + 1);
        }
        cout << endl << endl;

        for(i = 0; i < 9; i ++)  {
                for(j = 0; j < 9; j ++)
                        cout << setw(4) << tab[i][j];
                cout << endl;
        }

        return 0;
}
```

该函数的输出结果与例 3.14 完全相同。不同的是，在本例中，九九表被放在一个二维数组中保存了起来。

5.1.2.2 多维数组的初始化

与一维数组一样，多维数组也可以进行初始化。例如：

```
int    ia[2][2] = {{1, 2}, {3, 4}};
```

其中，内层的花括号并非必需的，对该数组的初始化完全可以改写成

```
int    ia[2][2] = {1, 2, 3, 4};
```

但习惯上，人们还是用花括号将每一维的初始化值分别括起来，以提高程序的可读性。

在带有初始化数据的多维数组说明中，最多只能省略其第一维中的常数，否则，编译器将由于无法了解各维的大小而导致编译失败。例如：

```
int    ia[][2] = {{1, 2}, {3, 4}};
```

就说明了一个 2×2 的二维数组。而

```
int    ia[][] = {{1, 2}, {3, 4}};
float  fa[][][2] = {{{1.2, 2.3}, {3.4, 4.5}}, {{5.6, 6.7}, {7.8, 8.9}}};
```

均属于错误的说明，尽管在初始化表中使用了花括号。

5.1.2.3 多维数组的存储形式

由于多维数组可以看作是一个元素为数组的数组，所以多维数组具有图 5-2 所示的存储形式。

从图可见，整个 iarr 是按一个一维数组的形式来存放它的各个元素 iarr[i] 的；而它的每一个元素 iarr[i]，由于本身也是一个数组，因此也是按一个一维数组的形式来存放其各个元素 iarr[i][j]的。从下标值的变化来看多维数组的存储形式，可以看出：第 1 维下标变化得最慢，最后一维下标变化得最快。

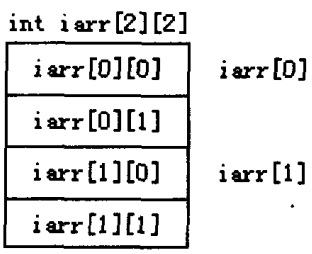

图 5-2 多维数组的存储形式

5.1.3 数组间的赋值

在 C++语言中，数组之间不能整体赋值。欲将一个数组赋给另一个数组，只有逐个元素地进行赋值。

例 5.3 数组间的赋值。

```
//      EX5_3.CPP
#include      <iostream.h>

int     main()
{
        int   arr[] = {1, 2, 3, 4, 5}, brr[5], i;

        for(i = 0; i < 5; i ++)
                brr[i] = arr[i];

        cout << "Array A\t";
        for(i = 0; i < 5; i ++)
                cout << arr[i] << '\t';
        cout << endl;
```

```
    cout << "Array B\t";
    for(i = 0; i < 5; i ++)
        cout << brr[i] << '\t';
    cout << endl;

    return 0;
}
```

该程序的输出为：

```
Array A    1    2    3    4    5
Array B    1    2    3    4    5
```

在进行数组间的赋值时，赋值结果的正确性必须由程序员来保证。其中最主要的就是要防止下标的超界。

例 5.4 赋值过程中数组超界。

```
//    EX5_4.CPP

#include    <iostream.h>

int    main()
{
    int    arr[] = {1, 2, 3}, brr[] = {4, 5, 6}, crr[5];

    for(int i = 0; i < 5; i ++)
        crr[i] = brr[i];

    cout << "Array C\t";
    for(int i = 0; i < 5; i ++)
        cout << crr[i] << '\t';
    cout << endl;

    return 0;
}
```

该程序的输出为：

```
Array C    4    5    6    1    2
```

由上例的输出可以看出，程序中不仅将数组 brr 的 3 个元素赋给了 crr 的前 3 个元素，而且还将数组 arr 的前两个元素赋给了 crr 的第 4，5 元素。这是由于在 for 循环中进行了 5 次赋值操作，当循环变量 i 的值为 3 时，已经超出了数组 brr 的上界。但 C++系统并不检查数组下标超界与否，它只是机械地将 brr[2]后边的两个字节（由于是 int 型）的内容赋给 crr[3]，根本不管这两个字节的内容是什么。往后的操作亦是如此。在本例中，由于 brr[2]的后边存放的正好是 arr[0]（一般情况下，C++编译器总是将局部变量按其说明时相反的顺序连续存放的，不管它们的类型是否相同），所以就会出现上述的结果。但应强调指出的是，该例只是为了说明 C++不做数组下标超界检查，读者尤其是初学者千万不要把它当作一种技巧。数

组下标超界的后果有时是极其严重的，甚至可能导致系统崩溃。

5.1.4 数组与函数

由于数组结构上的特殊性，使得数组作为函数的参数时存在两种调用形式：单个数组元素作为函数的参数和整个数组作为函数的参数。

5.1.4.1 单个数组元素作为函数的参数

由于数组的每个元素都是一个具有相同数据类型的变量，所以，从函数的角度来看，用单个数组元素作为函数的参数与普通变量作为函数参数是没有什么区别的。即，函数的形参与普通变量作为函数参数时没有什么变化，只是实参用的是下标变量而已。

由于普通变量作为函数参数属于赋值调用，所以用单个元素作为函数参数时，亦不会因函数对形参的修改而改变实参的值。

例 5.5 单个数组元素作为函数的参数。

```
//      EX5_5.CPP

#include      <iostream.h>

inline int      square(int x)
{
      return(x * x);
}

int     main()
{
      int     arr[5][2], i, j;

      for(i = 0; i < 5; i ++)
            arr[i][0] = i + 1;

      for(i = 0; i < 5; i ++)
            arr[i][1] = square(arr[i][0]);

      for(i = 0; i < 5; i ++)  {
            for(j = 0; j < 2; j ++)
                  cout << arr[i][j] << '\t';
            cout << endl;
      }

      return 0;
}
```

该程序的输出为：

```
1     1
2     4
3     9
4     16
```

5　25

5.1.4.2　整个数组作为函数的参数

从例 5.3 可以看出，用单个数组元素作为实参来调用函数其效率是很低的。C++语言允许将整个数组作为函数的参数。这时，函数应在其形参表中指明其参数是一个数组。

例 5.6　二次方程求根函数。

```
//      EX5_6.CPP

#include      <iostream.h>
#include      <math.h>

int   Quadratic(double factor[3], double result[2]);

int     main()
{
        double          f[] = {3, 6, 2}, r[2];

        if(Quadratic(f, r))
                cout << "The solution is: X1 = " << r[0] << '\t' << "X2 = " << r[1] << endl;

        return 0;
}

int   Quadratic(double factor[3], double result[2])
{
        double          delta;

        delta = factor[1] * factor[1] - 4 * factor[0] * factor[2];
        if(delta < 0)
                return 0;

        delta = sqrt(delta);
        result[0] = (-factor[1] + delta) / (2 * factor[0]);
        result[1] = (-factor[1] - delta) / (2 * factor[0]);

        return 1;
}
```

该程序的输出为

The solution is: X1 = -0.42265 X2 = -1.57735

函数 Quadratic（）带有两个参数：factor[3]为一具有 3 个元素的一维数组，这 3 个元素分别为方程的 3 个系数 a，b 和 c；result[2]用来存放方程的解 X1 和 X2。函数的值是一个标志：若为 0，则表示该方程无解。这时，result 中的数据无意义；若为 1，则说明 result 中的两个值就是方程的解。

应当说明的是，当用整个数组作为函数的参数时，对函数的调用属于地址调用。地址调用时，程序将实参的地址传给了形参。在函数中对形参的一切操作实际上是对它所对应之内存地址中的值进行的，因而就有可能改变实参的值，因为形参和实参表示的是内存中的同一

地址。例如实参 r，当它调用函数 Quadratic（）之前，其两个元素的值是没有意义的。在函数调用之后，如果方程有解，由于在函数中对形参进行了赋值，所以其两个元素的值就为方程的两个解。

为了防止地址调用时，在函数中由于偶然因素修改了实参的值，可以在函数的参数表中将不允许修改的参数说明为 const 的。例如：

```
int    Quadratic(const double factor[3], double result[2]);
```

这样，若在函数中出现对 factor[i]的赋值，编译器将会指出这一错误。

在函数参数表中的数组，允许省略其第一维的常数。例如：

```
int    Quadratic(const double factor[], double result[])
int    Quadratic(const double [], double []);                    // 仅函数原型
```

再如：

```
void  f(int [][3]);
```

但是：

```
void  f(int [][]);
void  f(int [2][]);
```

都将导致一个编译错误。

函数也可以返回一个数组的地址。由于这要涉及到指针的概念，对此将在第 6 章中介绍。

5.2 字符串

C++语言中有一种特殊的数组，它具有字符数组的形式，但却不仅是字符的集合，其内容往往表示的是有意义的句子。这种字符型数组就是字符串变量，简称字符串或串。

5.2.1 字符串的存储形式

一个字符串就相当于一个一维字符数组，它在内存中的存放形式与一维数组完全相同。但是，从内容（即元素的值）方面来看，它与一维字符数组最重要的区别在于：一个字符串中至少得有一个值为 0 的元素，在 C++语言中称其为 NULL 字符或空字符，通常用 '\0' 来表示，该元素标志着字符串的结束。而在一个字符型数组中，一般无此元素。例如：

```
char  ch[] = {'C', '+', '+'};
char  str[] = {'C', '+', '+', '\0'};
```

就将 ch 说明为一个字符型数组，它有 3 个元素，分别为字符'C'，'+' 和 '+'；而 str 则被说明为一个长度为 3 的字符串，其内容为 "C++"。可见，说明一个长度为 n 的字符串，其数组长度至少须为 n + 1。比如：

```
char  str[81];
```

就说明了一个最大长度为 80 的字符串。

与字符型数组不同的是：字符串不仅可以像字符数组那样逐个字符地进行初始化（注意，不要忘了字符串结束标志 '\0'），而且还可以用一个字符串常量来对其进行初始化。例如：

```
char  str[] = "C++ language";
```

说明了一个长度为 12 的字符串，这时，由于字符串常量中已包含了结束标志 '\0'，所以不再需要附加该字符。再如：

```
char  str[81] = "C++ language";
```

则说明了一个最大长度为 80 的字符串，且只对它前 12 个字符进行了初始化。

在对字符串进行初始化时，数组超界问题应由程序员负责。

5.2.2 字符串数组

可以说明这样一个数组：它的元素是一个个的字符串。这就是字符串数组。字符串数组实际就是一个二维字符数组。比如：

```
char  sarr[5][21];
```

就说明了一个包含有 5 个字符串，且每个字符串的最大长度为 20 的字符串数组。

与多维数值数组一样，字符串数组也允许进行初始化，例如：

```
char    sarr[3][21] = {"C++ language", "C language", "C++ is better then C"};
```

就在说明字符串数组 sarr 的同时对它的 3 个元素进行了初始化。

例 5.7 使用字符串。

```
//      EX5_7.CPP

#include      <iostream.h>

int     main()
{
        char    sarr[3][21] = {"C++ language", "C language", "C++ is better then C"};

        for(int i = 0; i < 3; i ++)
                cout << sarr[i] << endl;

        return 0;
}
```

该程序的输出为：

```
C++ language
C language
C++ is better then C
```

由例 5.7 的输出可以看出，在输出语句中，可以将字符串当作一个整体来操作。这一特点是普通数组所不具备的。然而，由于 sarr 只是一个数组，只不过它的元素是字符串而已。因此，对 sarr 的输出也只能一个一个元素地进行。

5.2.3 字符串间的赋值

因为字符串从本质上来讲是一个数组，所以字符串之间的赋值操作与数组一样，也只能逐个元素地进行。但由于字符串自身的特殊性，使得字符串之间的赋值操作要比普通数组来得方便。

例 5.8 复制字符串。

```
//      EX5_8.CPP

#include      <iostream.h>

int     main()
{
        char    str1[] = "C++ language", str2[20];
        int     i = 0;

        while(str1[i] != '\0')  {
                str2[i] = str1[i];
                i ++;
        }
        str2[i] = '\0';                                          // A

        cout << "String1: " << str1 << endl;
        cout << "String2: " << str2 << endl;

        return 0;
}
```

该程序的输出为：

```
String1: C++ language
String2: C++ language
```

程序中的 while 循环执行将 str1 复制给 str2 的操作。对比例 5.3 中将数组 a 赋给数组 b 的 for 循环可以看出，在复制一个字符串时，可以不必关心被复制字符串的长度。这是因为可以利用字符串的结束标志 '\0'，程序中的 while(str1[i] != '\0') 就是测试 str1[i]是否为结束标志，以确定循环是否结束。注意，程序中的行 A 是非常重要的，如果没有这条语句，将导致字符串 str2 没有正确的结束标志。

在进行字符串间的赋值时，也存在下标超界的问题，其突出表现为目标字符串的定义长度小于赋给它的字符串的实际长度。因此，程序员应保证作为目标字符串的字符数组应具有足够的长度，否则将会导致不可预料的后果。

5.2.4 字符串与函数

字符串虽是一个数组，但它本身却是一个整体，字符串中单个元素作为函数参数一般是没有什么实用意义的。因此，在函数间传递字符串时，一般都是以整个数组作为参数的。

例 5.9 字符串拷贝函数。

```
//      EX5_9.CPP
#include      <iostream.h>

int     str_copy(char str1[], const char str2[])
{
        if(!str2[0])
                return 0;                                          // A

        int     i = 0;

        do      {
                str1[i] = str2[i ++];
        } while(str2[i]);
        str1[i] = '\0';

        return 1;
}
```

函数 str_copy（）中的两个参数都是数组。由于整个数组作为函数参数时属于地址调用，所以，为了防止在函数中对源串的修改，这里将它说明为 const 的。函数的值表示拷贝操作是否成功。在该函数中，若源串为一空串，则返回失败标志 0，如行 A 所示。

在实用中，常将字符串参数说明成一个字符型指针，这一点将在第 6 章中介绍。

5.3 字符串处理库函数

C++系统为了方便用户的使用而提供了许多用于处理字符串的库函数，这些函数均在头文件 string.h 中说明。本节介绍几个常用的字符串处理函数。

1. **strlen（）函数** 函数原型：unsigned strlen(const char *str);

该函数用来求一个字符串的长度。它要求一个字符串作为它的参数，该字符串就是欲求其长度的字符串。参数 const char *str 是以指针形式给出的，这里，暂时将它理解为 const char str[]（下同）。函数返回一个无符号整数，它就是所求字符串的实际长度（不包括结束标志'\0'）。

2. **strcpy（）函数** 函数原型：char *strcpy(char *str1, const char *str2);

该函数将字符串 str2 的内容（包括结束标志）复制到 str1 中去，函数返回字符串 str1。用户在程序中可以测试该返回值是否为 0，以确定函数调用是否失败。操作失败的原因通常是由于 str1 与 str2 发生了重叠。比如，在说明这两个数组时 str1 正巧紧跟在 str2 后边且 str1 的长度又过小。

3. **strcat（）函数** 函数原型：char *strcat(char *str1, const char *str2);

该函数将字符串 str2 的内容连接到 str1 的后边，返回 str1。使用该函数时，程序员必须保证数组 str1 足够大且其中未使用的空间要能容纳下 str2 的内容。在连接过程中，str2 中的第一个字符代替了 str1 中的结标志，所以连接后的 str1 中只保留最后的那个结束标志。

4. **strcmp（）函数** 函数原型：int strcmp(const char *str1, const char *str2);

该函数用来比较字符串 str1 和 str2 的大小。其返回值为一整数，该数的值为负、零和正

分别表示 str1 小于 str2，str1 等于 str2 和 str1 大于 str2。所谓比较两个字符串的大小是指逐个地比较两个字符串中对应字符 ASCII 码值的大小，而不是比较它们长度的大小。比如：

```
strcmp("c", "C++")
```

返回值为 32，说明字符串"c"大于字符串"C++"。这是因为小写字母 c 的 ASCII 值为 99，而大写字母 C 的 ASCII 值为 67，从而比较的结果为前者大。再如：

```
strcmp("abcd", "abcde")
```

返回值为 - 101，说明第二个字符串大。其原因请读者自行分析。

例 5.10 使用字符串处理库函数。

```
//      EX5_10.CPP

#include      <iostream.h>
#include      <string.h>

int     main()
{
        char    s1[81] = "The C++", s2[] = "better then C", s3[10];

        cout << "The length of string1 is " << strlen(s1);
        cout << ", its content is: \" << s1 << "\" << endl;

        int     i = strlen(s2);

        cout << "The length of string2 is " << i;
        cout << ", its content is: \" << s2 << "\" << endl;
        cout << "The length of string3 is " << strlen(s3) << endl;
        strcpy(s3, s1);
        cout << "The length of string3 now is " << strlen(s3);
        cout << ", and its content is: \" << s3 << "\" << endl;

        int     sign = strcmp(s1, s2);

        if(sign < 0)
                cout << "String1 is less then string2\n";
        else
                if(sign == 0)
                        cout << "Two strings have same content\n";
                else
                        cout << "String1 is big then string2\n";

        strcpy(s3, " is ");
        cout << "String3 now has been changed to: \" << s3 << "\" << endl;
        strcat(strcat(s1, s3), s2);                                    // A
        cout << "The length of string1 now is " << strlen(s1) << endl;
        cout << "and its content is changed to: \" << s1 << "\" << endl;

        return 0;
}
```

该程序的输出为：

```
The length of string1 is 7, its content is: 'The C++'
The length of string2 is 13, its content is: 'better then C'
The length of string3 is 0
The length of string3 now is 7, and its content is: 'The C++'
String1 is less then string2
String3 now has been changed to: ' is '
The length of string1 now is 24
and its content is changed to: 'The C++ is better then C'
```

注意程序中的行 A，它等价于以下两条语句：

```
strcat(s1, s3);
strcat(s1, s2);
```

该行中利用了对函数 strcat（）第一次调用时的返回值（本例中是字符串 s1）作为对该函数第二次调用时的第一个参数。

对于像 strcat（），strcpy（）这样既将一个字符串作为它的参数又返回该字符串的函数，由于其参数属于地址调用，函数中所作的修改已经在实参中反应出来了，其返回值似乎有些多余。其实，这样的返回值是非常必要的：其一，可以利用该返回值来测试所要求的操作是否成功；其二，可以将对一个函数的调用用作另一个函数的实参，如例 5.10 行 A 所示。

习题

5-1 用一个一维数组存放自然数 1～10 的平方值，编写程序根据用户输入的整数值（1～10）输出其平方值。

5-2 定义一个可存放 30 个实数的一维数组来存放一个学生班（不超过 30 个人）某门功课的成绩。成绩由用户输入。设计一个循环，根据用户所输入的号码（1～30）输出对应学生的成绩。当用户输入 0 时循环结束，同时整个程序结束。

5-3 设计一个二维数组，存放一个学生班（不超过 30 个人）5 门功课的成绩及平均成绩。各门功课成绩由用户输入，平均成绩通过计算而得。设计一个循环，根据用户所输入的号码（1～30）输出对应学生的各门成绩与平均成绩。当用户输入 0 时循环结束，同时整个程序结束。

5-4 将习题 5-2 和 5-3 中的输入和输出操作写成函数的形式。

5-5 编写一个函数，对一个一维整型数组进行排序。数组中元素的个数由参数给出。

5-6 编写一个函数，实现 C++语言库函数 strlen（）的功能。

5-7 编写一个函数，实现 C++语言库函数 strcmp（）的功能。

5-8 编写一个函数，实现 C++语言库函数 strcat（）的功能。

5-9 编写一个递归函数，将一个字符串按其逆序输出。比如，设原串的内容为 abcdef，则输出为 fedcba。

第6章　指针和引用

指针是 C++语言最重要的特色之一。正确灵活地运用指针，可以有效地表示复杂的数据结构；能动态地分配内存；高效而方便地使用数组和字符串；可以使函数返回多于一个的值；能直接处理内存单元、访问硬件端口等。这些对设计系统软件都是非常必要的。正确熟练地掌握指针的应用，可以使所设计的程序简洁、紧凑、高效。然而，指针又是 C++语言中最危险的特性之一，错误地使用指针有可能导致系统的崩溃。在设计一个有一定规模的 C++程序时，不使用指针几乎是不可能的；使用不好其后果又是极其严重的。因此，在学习 C++语言时，深入地理解指针的本质、熟练地掌握它们的正确应用，将是至关重要的。

引用是 C++语言引入的一个与指针密切相关的概念，它具有与指针相似的功能。但从使用方面来讲，引用比指针显得更加自然且更为方便。

6.1　指针

欲正确地使用地指针，必须熟练地掌握指针的概念、对指针的访问以及指针的运算。

6.1.1　指针概念

数据在计算机中的存储方式是：按其所具有的数据类型，占据一定数量的连续内存单元，而每一个内存单元（可以存储一个字节即 8 位二进制数据）都对应一个唯一的编号，这就是该内存单元的“地址”。内存单元就好像旅馆中的房间，其中存放的数据就好像住在该房间中旅客，而地址就好像该房间的房号。“旅客”是会经常改变的，而“房号”则是恒定不变的。

指针变量——简称指针——是用来存放内存单元地址的变量，即指针的值不是一个普通的数据，而是一个地址值。比如，设一个指针的值为 2000，就是说明真正有效的数据是存放在地址号码为 2000 的内存单元中，如图 6-1 所示。可见，指针实际上是提供了一种对数据进行间接访问的手段。

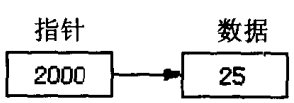

图 6-1　指针与内存地址

6.1.2　指针的说明

指针是变量，所以必须先说明才可以使用。指针说明的一般形式为

<storage> type　　*pointer;

其中：pointer 为所说明的指针变量名；星号“*”指出其后说明的变量是一个指针变量；而

类型 type 则指出所说明的指针变量是指向那种数据类型的指针。比如：

```
int    *ip;
char   *cp;
float  *fp;
```

就说明了 3 个指针 ip，cp 和 fp，它们分别为指向整型、字符型和实型数据的指针。

一般说来，内存地址值的长度值是固定不变的，通常，在 16 位环境中为两个字节、在 32 位环境中为 4 个字节。也就是说无论是 int 型、char 型还是 float 型的指针，其本身所占据的存储区域都是一样大的。那么，说明指针的类型到底有什么用呢？指针的类型主要有两个作用：其一，规定了用指针间接访问数据时的访问方式。比如，读取一个 char 型指针所指的数据时，只须读一个字节；而读取一个 int 型指针所指的数据时，则需要从该指针所指的内存单元以及下一个内存单元连续地读取两个字节（16 位操作系统）或 4 个字节（32 位操作系统）。其二，为指针的算术运算（将在 6.1.4.3 中介绍）提供依据。

在说明一个指针的同时也可以对它进行初始化，通常是用一个与所说明的指针具有相同类型之变量的地址来进行初始化的。例如：

```
int    i, *pi = &i;
```

就说明了一个整型变量 i 和一个整型指针 pi，同时将 pi 的值初始化为变量 i 的地址。运算符“&”称作取地址运算符，这是一个一元运算符，它的运算结果为其操作数的地址。取地址运算符要求其操作数必须是一个变量或对象。

下面对指针的初始化都会导致编译错误：

```
int    i;
char   c;
int    *pi = i;          // 用变量本身而不是变量的地址来初始化指针
int    *ip = &c;         // 用不同类型变量的地址来初始化指针
```

若将上述最后一行改为以下的形式，则属于合法说明：

```
int    *ip = (int*)&c;   // 使用强制类型转换
```

也可以用一个整型常量来初始化一个指针：

```
char   *cp = 0x80;
```

就使得指针 cp 指向地址编号为十六进制 80 的内存单元。但在这种情况下，程序员必须从硬件方面非常清楚该内存单元的作用，否则其后果将是极其严重的。因此，对初学者来讲，最好不要用这种方法来初始化一个指针。

6.1.3 对指针的访问

一个指针一经定义，就可以在程序中访问该指针。对指针的访问一般有两种形式：一是访问指针的值；一是访问指针所指内存单元中的数据。一般将后一种访问称为访问指针的内容。

对指针值的访问通常是用来将一个指针的值赋给另一个指针或是进行指针运算，一般情况下，程序员并不关心指针的具体值。例如：

```
int    i, *ip = &i, *iq;

iq = ip;
```

这就使得指针 ip 与 iq 指向同一内存单元，即变量 i 所占据的内存单元。

访问指针的内容时要用到递引用运算符“*”。递引用运算符又叫取内容运算符，它是一个一元运算符，要求一个指针作为它的操作数。该运算符的运算结果为其操作数的内容。例如：

```
int    i = 5, *pi = &i;

cout << *pi << endl;              // A  输出 5
*pi = 8;                          // 将 pi 的内容改为 8，相当于 i = 8
cout << i << endl;                // 输出 8
```

需要说明的是：当一个指针指向某一变量时，其值实际上是该变量的首地址。比如，这里的指针 pi，由于它是一个 int 型的指针，所以它所指的变量 i 就需要占据多个字节的内存单元，而在 pi 中存放的只是 i 的第一个字节的地址。当行 A 输出 pi 的内容（即 i 的值）时，系统会自动地根据指针的类型正确地取出变量 i 的值并将它输出到屏幕上。

应当强调指出的是：向一个未初始化的指针的内容进行赋值是极其危险的。当说明一个指针时，编译器并不对该指针作任何初始化工作，因此指针的值是一个随机值。该值所代表的内存单元可能是操作系统正在使用的一个关键的内存单元，若向这一单元写入一个值，就有可能导致系统的崩溃。所以，类似以下的语句是绝对不允许在程序中出现的：

```
int    *pi;

*pi = 5;                  // 指针 pi 的值不明确
```

也就是说，一个指针只有具有明确的值以后才可以对它的内容进行写访问。

6.1.4 指针的运算

指针的运算只有赋值运算、关系运算和算术运算 3 种。

6.1.4.1 指针的赋值运算

指针的赋值运算是指将一个地址值赋给一个指针。例如：

```
int    i, *p, *q;

p = &i;                   // 将变量 i 的地址赋给指针 p
q = p;                    // 将指针 p 的值，即变量 i 的地址赋给指针 q
```

指针的赋值运算一般只在类型相同的指针之间进行，若数据类型不同，须经强制类型转换后再进行赋值。比如：

```
int    i;
char   *pc;

pc = (char*)&i;
```

但是，在这种情况下程序员必须清楚为什么要进行这样的赋值。

注意：对指针的强制转换必须写成(type*)pointer 或(type*)(pointer)的形式，不得写成type*(pointer)的形式。

6.1.4.2 指针的关系运算

指针可以进行关系运算，两个指针的关系运算表示它们所指的变量在内存中的位置关系。比如，对于上例，则 p == q 的运算结果为真。

若 p < q 为真，则说明指针 p 所指的变量在内存中的地址号比 q 所指的变量在内存中的地址号小。这样的关系运算常用于对数组的访问中。

在返回指针的函数中，常返回一个值为 0 的指针来表示某种操作的失败。值为 0 的指针在 C++语言中称作空指针，它表示指针的一种状态。在程序中可以利用关系运算 p == NULL 来测试指针 p 是否为一空指针，以确定某种操作是否成功。这里的 NULL 是 C++预定义的一个宏

```
#define     NULL     0
```

应当指出的是：若某指针为一空指针，则绝对不允许对它的内容进行写访问。另外，一些早期的 C++编译器没有提供 NULL 宏，为了保持兼容，本书中的所有范例均用 0 表示空指针。

6.1.4.3 指针的算术运算

可以对指针进行算术运算。但在实用中，只对指针进行加减运算。例如：

```
int    i, *pi = &i;

pi += 5;
```

指针的算术运算与普通的数值运算不同，它并不是简单地将指针的值加上或减去给定的数值。对于算术运算 pointer ± n 的实际操作为

```
pointer ± sizeof(type) * n
```

比如，设在 16 位环境中，变量 i 的首地址即 pi 的值为 2000，则经过上运算后，pi 的值就变为 2010（而不是 2005）。可以看出，指针的类型对指针的运算起着相当关键的作用。

例 6.1 指针的使用。

```
//      EX6_1.CPP

#include      <iostream.h>

int     main()
{
        int     i = 100, *ip = &i, j;
```

```
    j = *ip + 1;                                    // A 相当于 j = i + 1
    cout << "i:" << '\t' << i << endl;              // 输出 i 的值
    cout << "*ip:" << '\t' << *ip << endl;          // 输出 ip 的内容
    cout << "j:" << '\t' << j << endl;              // 输出 j 的值
    cout << "&i:" << '\t' << &i << endl;            // 输出 i 的地址
    cout << "&j:" << '\t' << &j << endl;            // 输出 j 的地址
    cout << "ip:" << '\t' << ip << endl;            // 输出 ip 的值
    cout << "&ip:" << '\t' << &ip << endl;          // 输出 ip 的地址
    cout << "ip + 1:" << '\t' << (ip + 1) << endl;  // 输出 ip+1 的值

    return 0;
}
```

该程序的输出为：

```
i:      100
*ip:    100
j:      101
&i:     0xfff4
&j:     0xfff0
ip:     0xfff4
&ip:    0xfff2
ip + 1:         0xfff6
```

请读者对照源程序与程序的输出，仔细地体会指针的值、指针的内容、指针的运算等概念。注意，由于递引用运算符的优先级高于算术运算符，所以程序中行 A 是先对指针 ip 取内容，然后将该内容（100，即变量 i 的值）加 1 后赋给变量 j。

应当说明的是，该程序输出中的所有地址值在不同的环境下可能不同，但各值之间的关系将保持不变。

对指针也可以进行增量或减量运算，在对指针进行增量或减量运算时亦须注意运算符的优先级与结合性。比如：

```
j = * ++ ip;
```

该语句的意思是：先将指针 ip 的值加 1，然后将修改后的 ip 之内容赋给变量 j。即相当于

```
j = *(++ ip);
```

这是由于递引用运算符与增量运算符的优先级虽然相同，但它们的结合性都是右结合之故。相反，语句：

```
j = ++ *ip;
```

则等价于

```
j = ++(*ip);
```

即将指针 ip 的内容加 1 后赋给变量 j，而 ip 本身不变。

设 p 和 q 为同一类型的指针变量，还可以对它们实施如下的算术运算：

p - q

该运算的结果是一个整型值，它表示这两个指针指向的地址位置之间的数据个数。例如，设 p 和 q 都是 float 型指针，且它们的值分别为 2020 和 2000， 则上述运算的结果为 5（16 位环境），即说明它们之间存在 5 个实型数据。

6.1.5 多级指针

由于指针是一个变量，它本身需要占据一定的内存单元，所以可以说明一个指针指向该内存单元，这就是所谓的指向指针的指针。说明一个指向指针的指针的一般形式为

<storage> type **pointer;

说明一个指向指针的指针时需要使用两个星号。在 C++语言中，还可以用 3 个星号说明一个指向指针的指针的指针；用 4 个星号说明一个指向指针的指针的指针的指针；……。这些统称为多级指针，但多于两级的指针很少有实用价值。

例 6.2 使用指向指针的指针。

```
//      EX6_2.CPP

#include      <iostream.h>

int     main()
{
        int     i = 5, *pi = &i, **pp = &pi;

        cout << "Address of pi: " << pp << endl;
        cout << "Address of i: " << *pp << endl;
        cout << "Content of i: " << **pp << endl;

        return 0;
}
```

该程序的输出为：

```
Address of pi: 0xfff2
Address of i: 0xfff4
Content of i: 5
```

从例 6.2 的输出可见，用一个星号访问一个指向指针的指针时，得到的是一个地址，而用两个星号访问一个指向指针的指针时，才得到一个数值。这一规则可以推广到任意多级指针。

6.2 指针与数组

在 C++语言中，指针与数组有着极为密切的关系。可以用指针来访问一个数组的各个元素，也可以将一个数组说明成指针的形式。这不仅提高了程序设计的灵活性，同时也改善了

程序的运行效率。

6.2.1 用指针访问数组元素

在 C++语言中，一个数组名实际上就是一个指针，数组名所对应的就是数组中第一个元素的地址。然而，数组名相当一个 const 指针（将在 6.6.2 中介绍），因此不允许对数组名进行指针的赋值运算。

6.2.1.1 指向数组元素的指针

一个指向数组元素的指针实际上就是一个普通的指针，只不过通常它的初值为一个数组的首地址，即数组中第一个元素的地址。例如：

```
int    array[10];
int    *ap;

ap = &array[0];
```

这样，指针 ap 就成为一个指向整型数组 array 中元素的指针，它的初值为数组中第一个元素 array[0]的地址。由于数组名 array 中存储的就是元素 array[0]的地址，所以上述赋值语句可以改写为

```
ap = array;
```

在实际应用中，C++程序员更习惯于使用后者。上例还可以进一步简写为一行：

```
int    array[10], *ap = array;
```

6.2.1.2 通过指针访问数组元素

一个指针一旦指向了一个数组的某个元素，就可以通过该指针来访问数组中的各个元素。

例 6.3 利用指针访问数组元素。

```
//      EX6_3.CPP

#include      <iostream.h>
#include      <iomanip.h>

int     main()
{
        int     arr[10], *ap = arr, i;

        for(i = 0; i < 10; i ++)
                *(ap + i) = i + 1;                      // A

        for(i = 0; i < 10; i ++)
                cout << setw(4) << arr[i];
        cout << endl;

        return 0;
}
```

该程序的输出为：

1　2　3　4　5　6　7　8　9　10

例 6.3 的程序中，行 A 实际上是为数组 arr 中下标为 i 的元素赋值，它等价于语句

```
arr[i] = i + 1;
```

其执行过程为：先将指针 ap 加 i 个单位（本例中，一个单位为 sizeof(int)大小），使其指向数组 arr 中第 i 个元素的地址，然后将值 i + 1 以整型数据形式存入该地址（即将数据存入该地址所代表的存储单元中）。事实上，C++语言允许将一个指向数组元素的指针像下标变量一样的使用，例如，例 6.3 中的行 A 完全可以改写成以下的等价形式：

```
ap[i] = i + 1;
```

然而，在实用中很少以例 6.3 那样的方式来利用指针访问数组元素，因为这种方式与使用数组下标访问数组元素没有本质上的区别。由于数组在内存中是连续存放的，所以当顺序地访问一个数组时，通过对指针进行增减量运算来访问数组元素要比使用下标变量的效率高得多。

例 6.4　例 6.3 的另一版本。

```
//      EX6_4.CPP

#include      <iostream.h>
#include      <iomanip.h>

int     main()
{
        int     arr[10], *ap = arr, i;

        for(i = 0; i < 10; i ++)
                *ap ++ = i + 1;                         // A
        ap = arr;                                       // B

        for(i = 0; i < 10; i ++)
                cout << setw(4) << *ap ++;
        cout << endl;

        return 0;
}
```

该程序与例 6.3 的功能完全相同，但它的运行速度要比上例快得多，这是由于计算机处理增减量运算要比处理加减运算来得快。

应当指出的是，在例 6.4 中，行 B 的赋值是非常重要的。由于 C++的增减量运算本身就含有赋值意义，在程序的行 A 中对每个元素进行赋值时，同时也修改了指针 ap 的值。当赋值循环执行完毕时，指针 ap 实际上指向了并不存在的 arr[10]的地址。若不重新使 ap 指向 arr 数组的首地址，则其后的输出将是一些未知的值。

从以上的介绍可以看出，通过指针访问数组可以有 3 种形式：

```
*(ap + i)
ap[i]
*ap ++   或   *ap --
```

比较这 3 种使用形式：第一种突出了指针的使用；第二种显得最为自然；第三种的效率最高。在实用中，可以根据实际需要选用其中任何一个形式。在习惯上，最常用的是第二、三种形式。但应强调指出的是，第三种使用方式最为危险，实用中一定要特别小心。

6.2.2 指针与字符串

由于字符串本身就是一个数组，所以以上介绍的指针在数组中的应用同样适合于字符串。并且由于字符串自身的特点，使得指针在字符串中的应用显得更加方便。

例 6.5 例 5.9 的另一版本。

```
//      EX6_5.CPP

#include      <iostream.h>

int     str_copy(char str1[], char str2[])
{
        char    *cp1 = str1, *cp2 = str2;

        if(*cp2 == 0)                                   // A
                return 0;

        while((*cp1 ++ = *cp2 ++) != '\0');             // B

        return 1;
}
```

比较例 6.5 与例 5.9 可以发现，由于使用了指针，使得程序显得更加简练。注意：程序中的行 A 实际上是判断源字符串（str2）中第一个字符是否为字符串结束标志，若是，则没有必要进行复制，直接返回表示复制失败的 0 值。这里所采用的是从数值上进行判断——由于字符串结束标志 '\0' 字符的 ASCII 码值为 0，所以该关系表达式完全等效于*cp2 == '\0'。程序中的行 B 似乎有些不好理解，事实上，该语句完全等效于以下的 do-while 语句：

```
do    {
        *cp1 = *cp2;
        cp1 ++;
        cp2 ++;
} while(*cp1 != '\0');
```

由此可见，由于循环结束时源串中的字符串结束标志已经被复制到目标串中，所以在循环结束后无须再在目标串中添加一个字符串结束标志。

应当说明的是：尽管程序中的行 B 表现了一定的技巧，但却削弱了程序的可读性——while 循环中竟然没有循环体（仅为一条空语句）！因此，在实用中应尽量限制这种技巧性很强但可读性较差的编码方式。

6.2.3 指针数组和数组指针

指针是一个变量，所以可以说明其元素为指针的数组。同时，C++语言还允许说明指向数组的指针，使得利用指针进行数组的操作来得更加灵活。

6.2.3.1 指针数组

C++语言允许说明其元素为指针变量的数组，这种数组称作指针数组。说明一个指针数组的一般形式为

<storage> type *pointer_array[constant$_1$][constant$_2$]...[constant$_n$];

例如：

```
int    *pai[5];
char   *pac[3][2];
```

就分别说明了一个名为 pai 的一维整型指针数组和一个名为 pac 二维字符型指针数组。其中：数组 pai 包含有 5 个元素，每个元素都是一个整型指针；而数组 pac 包含有 6 个元素，每个元素都是一个字符型指针。

指针数组既具有数组的特点又具有指针的特点，可以像对数组的访问一样，通过下标变量来访问一个指针数组的元素，但这时访问到的是一个地址，只有对该地址取内容才可以访问到真正的数据。指针数组常和多维数组一起使用。

例 6.6 指针数组的使用。

```
//      EX6_6.CPP

#include      <iostream.h>

int     main()
{
        int     arr[3][2] = {{1, 2}, {3, 4}, {5, 6}} i, j;
        int     *pa[3];

        for(i = 0; i < 3; i ++)
                pa[i] = arr[i];                                  // A
        for(j = 0; j < 3; j ++)  {
                for(i = 0; i < 2; i ++, pa[j] ++)                // B
                        cout << "arr[" << j << "][" << i << "]=" << *pa[j] << '\t';
                cout << endl;
        }

        return 0;
}
```

该程序的输出为：

```
arr[0][0]=1  arr[0][1]=2
arr[1][0]=3  arr[1][1]=4
arr[2][0]=5  arr[2][1]=6
```

注意：程序行 B 中的 pa[j] ++是为了每循环一次使得指针指向下一个元素。当然，可以在 for 语句的控制部分中去掉该表达式，而将输出语句中的*pa[j] 改为 *pa[j] ++，则效果相同。

指针数组也可以在说明时进行初始化。比如，在例 6.6 中将指针数组 pa[3]的说明改为

```
int   *pa[3] = {arr[0], arr[1], arr[2]};
```

并删去程序中为数组 pa 赋值的 for 循环 A，则程序的功能不变。

在C++语言中，指针数组最常用的场合就是说明一个字符串数组。即说明一个数组，它的每个元素都是一个字符串。

例 6.7 字符串数组。

```
//      EX6_7.CPP

#include      <iostream.h>

int     main()
{
        char    *weekdays[] = {"", "Monday", "Tuesday", "Wednesday",
                                "Thursday", "Friday", "Saturday", "Sunday"};
        int     week;

        while(1)        {
                cout << "Enter a weekday number: ";
                cin >> week;
                if(week < 1 || week > 7)
                        break;
                cout << week << "-->" << weekdays[week] << endl;
        }

        return 0;
}
```

该程序反复地根据用户输入的星期日期输出它相应的英文名，当用户输入的日期小于 1 或大于 7 时，程序运行结束。

程序中对字符串数组 weekdays 进行初始化时将第一个字符串初始化为一个空串，其目的是为了不使用下标为 0 的数组元素，使得用户输入的日期与数组下标对应起来。这样的用法实际上只浪费了一个字节的存储单元，却改善了程序的运行效率。若在字符串数组中去掉了这一空串，则输出时必须将 weekdays[week] 改为 weekdays[week - 1]，这必然会增加程序的计算工作量。

应当说明的是：这样定义的字符串数组一般总是要进行初始化操作，或者使用动态内存分配技术（下面介绍）为每个指针赋上一个有效的地址，否则将有可能导致对未初始化的指针的访问。由于系统会为字符串常量自动地分配一块内存，当用一个字符串常量初始化一个指针时，指针的值即为该字符串常量的首地址。

当然可以将 weekdays 说明成一个二维字符数组的形式，但是如果那样做，不仅增加了说明时的复杂程度，而且还会造成内存单的浪费。比如：

```
char    weekdays[8][10] = {"", "Monday", "Tuesday", "Wednesday",
                           "Thursday", "Friday", "Saturday", "Sunday"};
```

这样，数组 weekdays 需要占据 80 个字节的内存单元。即使是省掉下标为 0 的空串，也需要有 70 个字节的存储单元。这是因为：为了能存放下其中最长的字符串“Wednesday”，至少需要 10 个字节。同时，用户在进行说明前，必须先找出所要存放的最长的字符串，以便确定数组的大小。而采用例 6.7 中的说明方式，编译器会自动地计算每个字符串的长度，并为它们分配相应的存储单元。实际上，例 6.7 中的字符串数组 weekdays 仅占据了 58 个字节的存储单元。

6.2.3.2 数组指针

C++语言还允许说明指向一个数组的指针，这样的指针叫作数组指针。说明一个数组指针的一般形式为

<storage> type (*arr_pointer)[constant$_1$][constant$_2$]...[constant$_n$];

例如：

```
int  (*ap)[5];
```

就将 ap 说明为一个指向包含有 5 个元素的整型数组的数组指针。在数组指针的说明中，圆括号是必须的，这是由于方括号“[]”较指针说明符“*”的优先级高，若无此圆括号，编译器将把上述说明解释成

```
int  *(ap[5]);
```

即包含有 5 个元素的指针数组。

例 6.8 例 6.6 的另一个版本。

```
//      EX6_8.CPP

#include      <iostream.h>

int     main()
{
        int     arr[3][2] = {{1, 2}, {3, 4}, {5, 6}}, (*ap)[2] = arr, *p, i, j;

        for(i = 0; i < 3; i ++, ap ++)  {                       // A
                for(j = 0, p = *ap; j < 2; j ++, p ++)          // B
                        cout << "arr[" << i << "][" << j << "]=" << *p << '\t';
                cout << endl;
        }

        return 0;
}
```

该程序的输出与例 6.6 完全相同。程序中在说明数组指针 ap 时对它进行了初始化，这实际上是将整型数组 arr[0][0]的地址赋给了它。程序行 A 中的 ap ++使得外循环每循环一次 ap 的值就加一个单位，这时一个单位的长度就不是一个整型数据的长度即两个字节，而是一个

包含有两个整型数据之数组的长度了。实际上，外循环每循环一次，就相继地将 arr[1][0]，arr[2][0]，…，的地址赋给了 ap。因为 ap 是一个数组指针，即它所指的是一个数组，而输出流 cout 不能将一个整型数组整个地进行输出，因此这里又说明了一个整型指针 p，程序中借助它来逐一地输出 ap 所指数组中的各个元素。

6.3 指针与函数

利用指针，可以更为方便地进行函数之间的数据交换以及更为灵活地调用一个函数。

6.3.1 指针作为函数的参数

指针可以用作函数的参数。当函数的某个参数是一个指针时，该参数就属于地址调用。即用实参调用函数时，系统就将实参的地址传给了函数，因此，函数中对形参内容的任何修改都将直接影响实参的内容。

例 6.9 地址调用对实参的影响。

```
//      EX6_9.CPP

#include      <iostream.h>

int     main()
{
        void    Swap(int*, int*);

        int     a = 3, b = 5;

        cout << "Before SWAP be called:\ta = " << a << "\tb = " << b << endl;

        Swap(&a, &b);                                                    // A

        cout << "After SWAP be called:\ta = " << a << "\tb = " << b << endl;

        return 0;
}

void    Swap(int *x, int *y)
{
        int     temp = *x;

        *x = *y;
        *y = temp;
}
```

该程序的输出为：

```
Before SWAP be called: a = 3      b = 5
After SWAP be called:  a = 5      b = 3
```

由上述输出可以看出地址调用时实参所受的影响。注意程序中行 A 的函数调用：由于

函数 Swap（）要求两个指针变量作为它的参数，而变量 a 和 b 均是普通的整型变量，因此必须取它们的地址做实参，以与函数的形参相匹配。

在第 5 章中曾介绍过，将整个数组传递给函数时属于地址调用。从本质上来讲，将一个数组传递给函数时，实际上就是将数组的首地址作为参数传给了函数。加之数组与指针有着密不可分的联系，因此在实用中，当需要将一个数组、一个字符串传给函数时，习惯上都将参数写成指针的形式。第 5 章中所介绍的字符串处理库函数就是采用的这种形式。

例 6.10 例 5.6 中函数 Quadratic（）的另一版本。

```
//      EX6_10.CPP

#include      <math.h>

int     Quadratic(double *factor, double *result)
{
        double delta;

        delta = factor[1] * factor[1] - 4 * factor[0] * factor[2];
        if(delta < 0)
                return 0;

        delta = sqrt(delta);
        result[0] = (-factor[1] + delta) / (2 * factor[0]);
        result[1] = (-factor[1] - delta) / (2 * factor[0]);

        return 1;
}
```

比效例 5.6 与例 6.10 可以发现：在两个版本中，函数的定义（即函数体）完全一样，它们的差别只是参数的书写方式不同。

在有些应用中，尤其是涉及到字符串的应用，用指针作为函数的参数还可以提高函数的运行效率。

例 6.11 例 5.9 的又一版本及对函数 Str_Copy()的演示。

```
//      EX6_11.CPP

#include      <iostream.h>

int     main()
{
        int     Str_Copy(char*, const char*);

        char    s1[20] = "The first", s2[20] = "The second";

        cout << "Address of s1: " << (int*)s1 << "\tContent: " << s1 << endl;
        cout << "Address of s2: " << (int*)s2 << "\tContent: " << s2 << endl;

        Str_Copy(s1, s2);

        cout << "Address of s1: " << (int*)s1 << "\tContent: " << s1 << endl;
```

```
    cout << "Address of s2: " << (int*)s2 << "\tContent: " << s2 << endl;

    return 0;
}

int    Str_Copy(char *str1, const char *str2)
{
    if(*str2 == '\0')
        return 0;

    while(*str2 != '\0')  {
        *str1 = *str2;
        str1 ++;
        str2 ++;
    }

    return 1;
}
```

该程序的输出为：

```
Address of s1: 0xffe2    Content: The first
Address of s2: 0xffce    Content: The second
Address of s1: 0xffe2    Content: The second
Address of s2: 0xffce    Content: The second
```

从例 6.11 可以看出：由于是地址调用，在函数 Str_Copy（）中将字符串 str2 的内容赋给了字符串 str1，从而导致实参 s1 的内容发生了变化；然而，在函数 Str_Copy（）的 while 循环中修改了指针 str1 和 str2 的值，为什么实参 s1 和 s2 的首地址在函数调用前后没有发生变化呢？这是由于地址调用时，实参将其值（一个地址）传给了形参，即对于上例，在进行函数调用时，等效地存在以下的隐含赋值操作：

```
str1 = s1;
str2 = s2;
```

以后，实参与形参就脱离了联系。也就是说，地址调用时，从参数值的角度来看仍是赋值调用。当函数对形参的内容进行修改时，因为是对同一地址中的内容进行的修改，因此就导致实参的内容发生了变化。这就是地址调用的实质。

程序的输出语句中对指针 s1 和 s2 进行强制类型转换是为了能输出字符串 s1 和 s2 的首地址，如果不进行这样的强制转换，cout 输出的将是字符串的内容。这是因为预定义的插入操作将一个字符型指针操作数当作一个字符串来处理的。

在设计一个参数可变的函数时，利用一个字符指针将一个指示参数个数与各参数类型的字符串传给函数，将使得函数变得更加灵活多变。例 6.12 中函数 Print（）就是对 C 语言的库函数 printf（）（C++程序也可以调用）的一个简单模拟。

例 6.12 一个简化了的 printf（）函数。

```
// EX6_12.CPP

#include    <iostream.h>
```

```
#include    <stdarg.h>

int    main()
{
       int    Print(char*...);

       char   *str = "C++ language";
       float  f = 34.5;

       Print("%s, %d, %c, %%%f\n", str, 25, 'c', f);                    // A

       return 0;
}

int    Print(char *format...)
{
       va_list ap;
       char   ch;
       int    i = 0;

       va_start(ap, format);
       while((ch = *format ++) != '\0')      {
              i ++;
              if(ch != '%')
                     cout << ch;
              else
                     switch(ch = *format ++)      {
                            case '%':
                                   cout << '%';

                                   break;

                            case 's':
                            {
                                   char   *var = va_arg(ap, char*);

                                   cout << var;
                            }

                                   break;

                            case 'd':
                            {
                                   int    var = va_arg(ap, int);

                                   cout << var;
                            }

                                   break;

                            case 'f':
                            {
                                   double         var = va_arg(ap, double);
```

```
                cout << var;
            }
                break;
            case 'c':
            {
                char    var = va_arg(ap, char);
                cout << var;
            }
                break;
        }
    }
    va_end(ap);

    return i;
}
```

该程序的输出为

C++ language, 25, c, %34.5

函数 Print（）的第一个参数是一个字符串，叫作格式说明串。在该串中，百分号“%”与它后边的一个叫作格式说明符的字符一起指出了与其对应的参数之类型。本例中，格式说明符只有 5 个：“s”对应一个字符串；“d”对应一个整型数据；“c”对应一个字符型数据；“f”对应一个实型数据；而“%”则对应百分号字符本身。也就是说，格式串中的每一个带有字母的百分号都对应一个缺省了的相应类型的参数，有多少个这样的百分号就有多少个缺省了的实参，调用函数时第一个实参就是通过这种方式告知函数实参的个数和它们的类型的。比如在行 A 中：%s 对应实参 str；%d 对应实参 25；%c 对应实参 'c'；%f 对应实参 f。格式串中其他字符按原样输出，比如行 A 中的逗号、空格和新行符。

在 Print（）函数的函数体中，while 循环通过移动字符指针逐一地读取格式串中的每一个字符，并根据每个字符的具体内容作出相应的反应。

注意：函数中每个 case 的花括号是必须的，这是为了节省内存，为每一个分支都说明了一个块作用域变量，这样，从 while 循环的块作用域中看，仅多说明了一个变量。如果没有花括号，由于这些变量的作用域将遍及整个 while 循环，则在循环块作用域中将要多说明 3 个变量。正因为如此，所有分支中说明的变量才可以取同一个名子 var。

6.3.2 返回指针的函数

指针可以用作函数的值。一个函数只能返回一个值，而不像函数的参数那样可以有多个，这样对有些应用来讲就显得不方便。然而，C++语言允许函数的返回值是一个指针，这就可以使函数返回像数组、字符串这样包含有多个数值的数据类型。

例 6.13 例 5.6 的另一版本。

```
//      EX6_13.CPP
```

```
#include    <iostream.h>
#include    <math.h>

int    main()
{
       double *Quadratic(double*);

       double f[3], *r;

       cout << "Please enter three factors: ";
       cin >> f[0] >> f[1] >> f[2];

       r = Quadratic(f);

       if(r)                                    // A
              cout << "The solution is: X1 = " << r[0] << '\t' << "X2 = " << r[1] << endl;
       else
              cout << "No solution!\n";

       return 0;
}

double *Quadratic(double *factor)
{
       static double          result[2];
       double         delta;

       delta = factor[1] * factor[1] - 4 * factor[0] * factor[2];
       if(delta < 0)
              return 0;                         // B

       delta = sqrt(delta);
       result[0] = (-factor[1] + delta) / (2 * factor[0]);
       result[1] = (-factor[1] - delta) / (2 * factor[0]);

       return result;
}
```

比较例 6.13 和例 5.6 中对函数 Quadratic（）的调用，可以看出前者显得更加自然。

应当说明的是：函数 Quadratic（）中的数组 result 是用来保存二次方程的两个根的，函数的返回值就是该数组的首地址。由于该数组是一个局部变量，若将它说明成自动的，则当函数 Quadratic（）的调用结束后，该数组将自然消失，它所占据的内存单元将返还给系统。这时，该数组所对应的内存中的内容将是一些不可预知的数据，一般情况下不再是函数中计算出的二次方程的两个根的值，因为系统很可能将函数返还给它的内存单元另作它用。因此，在一个函数中，若某个变量的首地址需要被函数返回，则该变量必须被说明为静态的或是利用运算符 new 动态分配的内存（将在 6.4 中介绍），以保证所返回数据的正确性。另外，程序的 main（）函数中用来接受函数 Quadratic（） 返回值的变量 r 必须说明成一个指针，且

其类型也应为 double。不能将 r 说明成一个数组，因为数组名相当一个常指针，常指针是不允许被重新赋值的。

函数 Quadratic（）中行 B 的作用是当方程无解时向主调函数返回一个空指针，以供主调函数检测用。main（）函数中的行 A 就是检测函数的返回值是否为空指针；该语句就相当于 if(r != 0)。

返回指针的函数更多地用来返回一个字符串，C++语言字符串处理库函数大都采用这种形式。

例 6.14 例 6.7 的另一版本。

```
//      EX6_14.CPP

#include      <iostream.h>

char    *WeekDays(int);

int     main()
{
        int     week;

        while(1)        {
                cout << "Enter a weekday number: ";
                cin >> week;
                if(week < 1 || week > 7)
                        break;
                cout << week << "-->" << WeekDays(week) << endl;
        }

        return 0;
}

char    *WeekDays(int n)
{
        static char     *wd[] = {"", "Monday", "Tuesday", "Wednesday",
                                "Thursday", "Friday", "Saturday", "Sunday"};

        return wd[n];
}
```

例 6.14 中的 WeekDays（）函数就显得比较通用。

6.3.3 C++语言中的 main（）函数

我们在使用 DOS 命令 DIR 时，可以这样用：

dir<CR>

也可以这样用：

dir *.*<CR>　　或者　　**dir filenam.ext<CR>**

其中，用空格与命令隔开的内容属于该命令的参数，叫作命令行参数。在 DOS 的命令集中，有些命令还可以带有多个命令行参数，比如：

copy file1.cpp file2.cpp<CR>

就带了两个命令行参数 file1.cpp 和 file2.cpp，各参数之间用空格隔开。

用C++语言是否可以写出具有类似形式的程序呢？回答是肯定的，那就是使用带参数的 main（）函数。

事实上，在C++语言中，预说明的 main（）函数原型为

int main(int argc, char *argv[], char *env[]);

即，main（）函数是一个值为整型量的函数，它可以带有多达 3 个参数：一个为整型量，其它两个为字符串数组。整型参数 argc 用来统计命令行参数的个数；字符串数组 argv 中存放用户输入的命令行参数；而字符串数组 evn 中存放的则是 DOS 环境变量。注意：第三个参数在有的系统中可能不能使用。

例 6.15 带参数的 main（）函数和命令行参数。

```
//      EX6_15.CPP

#include      <iostream.h>

int     main(int argc, char *argv[], char *evn[])
{
        if(argc == 1)  {
                cout << "\aYou forget entering command-line arguments.\n";

                return -1;
        }

        int     i;

        cout << "You have entered " << argc << " command-line arguments.\n\n";
        for(i = 0; argv[i]; i ++)                                          // A
                cout << "Argv[" << i << "]:\t" << argv[i] << endl;
        cout << endl;

        for(i = 0; evn[i]; i ++)                                           // B
                cout << "Evn[" << i << "]:\t" << evn[i] << endl;
        cout << endl;

        return 0;
}
```

设运行程序 EX6_15 时在 DOS 提示符下输入如下：

C:>ex6_15 This program demonstrates command-line arguments<CR>

则程序的输出为：

You have entered 7 command-line arguments.

```
Argv[0]:    C:\BORLANDC\USER\EX6_15.EXE
Argv[1]:    This
Argv[2]:    program
Argv[3]:    demonstrates
Argv[4]:    the
Argv[5]:    command-line
Aegv[6]:    arguments

Evn[0]:     COMSPEC=C:\COMMAND.COM
Evn[1]:     PROMPT=$p$g
Evn[2]:     PATH=C:\DOS;C:\WINDOWS;C:\BORLANDC\BIN;C:\CPAV
Evn[3]:     TEMP=C:\DOS
```

其中最后 4 行输出内容因机器设置不同而异。

若某个命令行参数中包含有空格，则应将该命令行参数用双引号括起来。比如，上述的输入可改为

C:>ex6_15 "This program demonstrates command-line arguments"<CR>

则程序的输出将变为：

```
You have entered 2 command-line arguments.

Argv[0]:    C:\BORLANDC\USER\EX6_15.EXE
Argv[1]:    This program demonstrates the command-line arguments

Evn[0]:     COMSPEC=C:\COMMAND.COM
Evn[1]:     PROMPT=$p$g
Evn[2]:     PATH=C:\DOS;C:\WINDOWS;C:\BORLANDC\BIN;C:\CPAV
Evn[3]:     TEMP=C:\DOS
```

从该程序的输出可以看出，系统会自动地根据用户键入的命令行参数填写各参数的内容。对应上述的第一个输入，系统统计出用户共键入了 7 个命令行参数（包括命令本身），并将值 7 赋给了 argc；同时将各个命令行参数顺序地赋给了 argv[0]～argv[6]，这时 argv[7]为一空串，才使得循环 A 得以终止；系统又将 DOS 的环境变量逐一地赋给了 evn[0]～evn[3]（这里假设用户所使用的计算机只设置了这 4 个环境变量），空串 evn[4]使得循环 B 得以结束。对应上述第二个输入，系统则给 argc 赋一值 2，并使得 argv[2]为一空串。实际上，C++语言并没有规定可以有几个命令行参数串和环境串，一个程序中可以出现几个这样的串完全是由程序运行时动态决定的。最多可以产生多少个命令行字符串仅受 DOS 的一个命令行不得超过 128 个字符的限制；最多可以有多少个环境串也仅受 DOS 环境变量设置的制约。这些已超出了本书的内容，这里不做讨论。

应当说明的是：在任何情况下，argc 的值至少为 1，即 argv[0]永远不会是一个空串。这是因为用户在运行一个程序时起码要键入程序名，即命令行起码得有一个参数。这就是说，argv[0]的内容永远是用户所执行之程序的程序名。但该字符串的内容与用户所键入的命令行

参数不完全相同。第一，该串中的字符全是以大写的形式出现的；第二，系统会自动地加上该程序在磁盘中的路径和文件的括展名。如上述的输出所示（这里假设程序存储在C:\BORLANDC\USER 子目录下）。

main（）函数的返回值可供系统来检查程序是否正常地结束。这在使用 DOS 的批处理命令顺序地执行多个程序时是非常有用的。习惯上，以返回 0 表示程序正常结束；而返回非 0 则表示程序非正常结束。在实用中，可以规定返回的非 0 值从数值上表示程序是由于什么原因而异常结束的。

利用带参数的 main（）函数可以使得一些程序使用起来显得更加自然。比如，编写一个计算阶乘的程序，使得用户在键入程序名的同时就可以键入所要计算的数，这就使得用户在使用计算机时就好象在使用一个计算器。

例 6.16 计算一个数的阶乘。

```
//      EX6_16.CPP

#include      <iostream.h>
#include      <stdlib.h>

long    Facto(int);

int     main(int argc, char *argv[])
{
        if(argc != 2)   {
                cout << "Usage: EX6_15 n<CR>" ;

                return 1;
        }

        cout << Facto(atoi(argv[1])) << endl;

        return 0;
}

long    Facto(int n)
{
        if(n == 0)
                return 1;
        return (n * Facto(n - 1));
}
```

对于该程序，用户可以这样输入：

C:>ex6_16 5<CR>

程序将输出 120 并结束运行。

例 6.16 中用到了一个 C++的库函数 atoi（），该函数的原型在头文件 stdlib.h 中被说明为

```
int     atoi(const char *str);
```

它的作用是将字符串 str 转换成一个 int 型的整数。调用该函数时，实参字符串必须具有数值

的形式，比如："1234"，"-5678" 等。实参也可以具有实数形式，但这时小数点及其以后的部分将不被转换。比如：

```
atoi("23.5")
```

的值为 23。

对于 main（）函数，以下的调用均是合法的：

```
main()                          // 不带任何参数，最基本的调用方式
main(int)                       // 没有什么实用意义，几乎从不这样调用
mian(int, char*[])              // 典型的带参数调用方式，几乎总是这样调用
main(int, char*[], char*[])     // 只在一些特殊场合中才这样调用
```

如果带有参数，各参数的顺序必须是：argc，argv，evn。需要说明的是：在定义 main（）函数时，参数名并非必须是 argc，argv 和 evn。这 3 个参数可以是任何合法的变量名，但习惯上总是使用这 3 个变量名。

当定义一个函数时没有给出函数值的类型时，系统默认该函数的值为 int 型。这时，与说明成返回值为 int 型一样，函数体中必须存在 return 语句，否则编译器将会给出一个警告信息。若不需要 main（）函数返回任何值，则必须明确地将它定义成值为 void 型的函数。实际上，大多数应用程序都不需要返回什么值，这时将 main（）函数说明成一个 void 型还可以省去函数体中的 return 语句，本书在以后的范例中均采用这样的写法。

6.3.4 指向函数的指针

在C++语言中，指针除了可以指向一个变量外，还可以指向一个函数，这就是所谓的指向函数的指针，简称函数指针。

6.3.4.1 说明函数指针

说明一个函数指针应具有以下的一般形式：

```
<storage> type    (*pointer)(<arg_list>);
```

其中：type 为该指针所指函数之返回值的类型；arg_list 为该指针所指函数的形参列表。例如：

```
int    (*funp)(int, char*);
```

就说明了一个名为 funp 的函数指针，该指针可以用来指向任何一个值为整型量的、带有两个类型分别为 int 型和 char*型参数的函数。

说明一个函数指针时，包围指针名的一对圆括号是一定不能省略的，比如：

```
int    *funp(int, char*);
```

说明的是一个返回值为整型指针的函数，而不是一个函数指针。

6.3.4.2 函数指针的运算

函数指针的运算只有赋值运算。

函数指针与普通指针一样，都代表着一个地址，函数指针所代表的地址是其所指函数在

内存中存放时的首地址，即函数的入口地址。函数指针的赋值运算就是将一个代表函数入口地址的值赋给函数指针

一个函数必须存储在内存中才可以被调用，而函数名就代表函数的入口地址。设有函数

```
int    fun(int, char*);
```

则赋值运算

```
funp = fun;
```

就使得函数指针 funp 指向了函数 fun（）的入口地址。

一个函数只能将它的入口地址赋给与其类型相同的函数指针。这里所讲的类型相同是指：不仅函数值的类型应与函数指针的 type 相同，而且函数的参数也必须与函数指针的 arg_list 一一对应。另外，运算中赋值运算符的左操作数只允许是函数指针，而不得是函数名，这是因为函数名相当一个 const 函数指针。

函数指针的赋值运算不仅在函数指针与函数名之间进行，而且还可以在函数指针之间进行，设 funq 是一个与 funp 类型相同的函数指针，则

```
funq = funp;
```

使得 funq 与 funp 取得相同的地址值，即指向同一个函数。

6.3.4.3 利用函数指针调用函数

利用函数指针调用函数，其形式与函数调用完全相同。

例 6.17 例 3.5 另一版本。

```
//      EX6_17.H

#if !defined   _EX6_17_H_
#define        _EX6_17_H_

int   Add(int, int);
int   Sub(int, int);
int   Mult(int, int);
int   Dev(int, int);

#endif


//      EX6_17.CPP

#include      <iostream.h>
#include      <stdlib.h>
#include      "ex6_17.h"

void   main(int argc, char *argv[])
{
       int     (*funp)(int, int);

       if(argc != 4)   {
```

```
            cout << "Usage: EX6_17 expression<CR>. For example:\n";
            cout << "EX6_17 123 + 456<CR>\n";

            exit(1);
        }

        int     err = 0;

        switch(argv[2][0])    {
            case '+':
                funp = Add;          // 将函数 Add()的入口地址赋给指针 funp

                break;

            case '-':
                funp = Sub;          // 将函数 Sub()的入口地址赋给指针 funp

                break;

            case '*':
                funp = Mult;         // 将函数 Mult()的入口地址赋给指针 funp

                break;

            case '/':
                funp = Dev;          // 将函数 Dev()入口地址赋给指针 funp

                break;

            default:
                cout << "Operator you entered is illegal;\n";
                err = 1;
        }

        if(!err)
            cout << funp(atoi(argv[1]), atoi(argv[3])) << endl;      // A
}


//      EX6_17A.CPP

#include      <iostream.h>

int     Add(int x, int y)
{
        cout << x << " + " << y << " = ";

        return x + y;
}

int     Sub(int x, int y)
{
```

```
    cout << x << " - " << y << " = ";

    return x - y;
}

int    Mult(int x, int y)
{
    cout << x << " * " << y << " = ";

    return x * y;
}

int    Dev(int x, int y)
{
    cout << x << " / " << y << " = ";

    return x / y;
}
```

该程序被写成 3 个文件：EX6_17.H，EX6_17.CPP 和 EX6_17A.CPP。在编译时可以将文件 EX6_17.CPP 和 EX6_17A.CPP 分别编译，然后连接成一个可执行文件 EX6_17.EXE。也可以利用做工程将 EX6_17.EXE 一次生成。

设运行该程序时的输入为

C:>ex6_17 123 + 456<CR>

则程序的输出为

123 + 456 = 279

这时，由于命令行中第三个参数为“+”，则程序就在 switch 分支中将函数 Add（）的入口地址赋给了函数指针 funp，于是行 A 中的 funp(atoi(argv[1]), atoi(argv[3])) 就相当于

Add(123, 456)。

若将命令行中的“+”换成“*”，则行 A 就相当于

cout << Mult(123, 456) << endl;

注意：使用本程序时，命令行中表达式的第一个操作数、运算符和第二个操作数之间必须用空格隔开，这是因为需要在 main（）函数中将它们分别解释为独立的命令行参数。

既然使用函数指针的目的是调用函数，那么为什么不直接调用函数，而是将函数的地址赋给函数指针，然后再用函数指针去调用函数呢？实际上，C++语言引入函数指针这一数据类型最主要的目的是为了能将函数作为函数的参数，具体例子参见下面的例 6.18。

6.3.4.4 函数指针数组

与普通指针变量一样，C++也允许说明元素为函数指针的数组——函数指针数组。函数指针数组可以是一维的，也可以是多维的，但实用中很少用到一维以上的函数指针数组。

说明一个一维函数指针数组的一般形式为

<storage> type   (*pointer_array[constant])(<arg_list>);

例如：

```
char *(funp_arr[10])(void);
```

就说明了一个包含有 10 个元素的函数指针数组，其中每个元素都是一个指向值为字符指针、无参数的函数之函数指针。

在说明一个函数指针数组的同时，可以对其进行初始化。实始化函数指针数组的初值必须为可见的函数名。

例 6.18 例 3.4 的又一版本。

```
//      EX6_18.CPP

#include      <iostream.h>
#include      <stdlib.h>
#include      "ex6_17.h"

void    main(int argc, char *argv[])
{
        int     Pos(const char, const char*);
        void    Out_Result(int, int, int (*)(int, int));

        char    *oper = "+-*/";
        int     (*funp[4])(int, int) = {Add, Sub, Mult, Dev};           // A

        if(argc != 4)   {
                cout << "Usage: EX6_17 expression<CR>. For example:\n";
                cout << "EX6_16 123 + 456<CR>\n";

                exit(1);
        }

        int     i = Pos(argv[2][0], oper);

        if(i < 0)
                cout << "Operator you entered is illegal\n";
        else
                Out_Result(atoi(argv[1]), atoi(argv[3]), funp[i]);      // B
}

int     Pos(const char c, const char *str)
{
        int     i = 0;

        while(*str)     {
                if(c == *str ++)
                        return i;
                i ++;
        }

        return -1;
}
```

```
void    Out_Result(int a, int b, int (*f)(int, int))
{
        cout << f(a, b) << endl;
}
```

该文件应与文件 EX6_17A.CPP 连接起来生成 EX6_18.EXE。

例 6.18 中的函数 Out_Result（）就是一个包含有函数指针参数的函数。注意：说明和定义一个含有函数指针参数的函数时，必须完整地给出指针的类型；而作为实参的函数指针则只须给出指针名，如程序中行 B 所示。作为实参的函数指针一定不能带有圆括号与其中的参数，否则编译器将把它解释成：先调用指针所指的函数，然后将其返回值作为实参。这将会导致一个错误。

程序中的行 A 是对指针数组 funp[]的说明，并对它进行了初始化。请读者注意它的初始化方式。程序中的函数 Pos（）和字符串 oper 是为了提高程序处理的自动化、简化程序的结构而设立的，其作用和执行过程请读者自行分析。

6.4 指针和动态内存分配

动态内存分配是指在程序运行期间，为程序中所需的、其大小不可预知的内存需求分配内存的一种方法。

C++语言属于编译性语言，而编译性语言要求程序中的变量必须事先说明，以使得编译器在程序运行之前就能够预先为每个变量分配相应的内存单元。变量在程序运行期间既不能添加，也不能减少。这在许多应用中是不方便的。

C++语言的动态内存分配机制为克服这种不便提供了手段。动态内存分配要用到指针、运算符 new 和 delete。

运算符 new 用来申请所需的内存，它的使用形式为

pointer = new type;

其中：pointer 为一 type 型的指针。new 运算符从一个称为堆的空闲内存中为程序分配一块与 type 相适应的内存单元，若分配成功，则返回指向这块内存首地址的指针；否则返回一个空指针。

运算符 delete 用来将申请到的内存返还给堆，它的使用形式为

delete pointer;

其中：pointer 应为一个保存着由 new 所申请之内存首地址的指针。

例 6.19 动态内存分配。

```
//      EX6_19.CPP

#include      <iostream.h>
#include      <stdlib.h>

void    main()
```

```
{
    int     *pi;

    cout << "Before allocation the value of pi is: " << pi << endl;
    pi = new int;
    if(pi == 0)     {
        cout << "Memory allocation failure!\n";

        exit(1);
    }

    *pi = 5;                                    // A
    cout << "After allocation the value of pi is: " << pi;
    cout << ", and its content is: " << *pi << endl;

    delete pi;                                  // B
}
```

当该程序运行正常时，其输出为：

```
Before allocation the value of pi is: 0x03ae
After allocation the value of pi is: 0x0d48, and its content is: 5
```

输出中，指针 pi 的原始值 3ae 是一个随机值。当内存分配成功后，pi 的值变为 d48，这就是所分配的内存单元的首地址。程序中的行 A 是向该内存单元中写入整数 5，这时，就好像程序是存在一个无名的整型变量，整型指针 pi 指向了该无名变量。行 B 的作用是将 pi 所指的、由 new 所分配的内存块返还给系统。通常，对应所有的 new 运算都必须有相应的 delete 操作，以确保将所有动态分配的内存统统返还给操作系统。这是程序员的责任。

应当说明的是：动态内存分配的操作完全后，应检查指针的值是否为 0，以确定内存需求申请是否成功。若内存分配失败，则对指针内容写访问的后果将可能是极其严重的。程序中的 if 语句就是进行这种检查，并作出相应的处理。为简单起见，在本书以后所举的例子中不再作这一类的检查。这一点请读者注意。

动态分配到的内存通常存放的是一些随机值，系统不对它们作任何初始化工作。因此，对这些值的读访问通常是无意义的，可以用 new 进行内存分配的同时为其指定初值。例如：

```
pi = new int(5);
```

就为所申请的内存单元指定了初值 5。若内存分配成功，则其中的内容就是整数 5。

在为动态分配内存指定初值时，圆括号中的内容可以是一个类型相同的变量或表达式，甚至可以是一个函数调用。比如，设整型变量 i 的值为 5，则下面的语句与上述语句等价：

```
pi = new int(i);
```

运算符 new 也可以为数组分配内存。为数组分配内存的一般形式为

pointer = new type[size][constant$_2$][constant$_3$]...[constant$_n$]

其中：size 为一整型表达式，其余的 constant$_i$ 必须为整型常量。

利用运算符 new 为数组动态分配内存时应注意：

1. 在为一个多维数组申请内存时，除了其第一维的大小可以用一个表达式来表示外，其他各维的大小必须用常量明确地给出。同时，接受一个多维数组的指针也应说明成相应的多维数组指针：

type (*pointer)[constant$_2$][constant$_3$]...[constant$_n$]

2. 不允许为动态分配的数组指定初值。比如，以下的初始化属于非法操作：

```
int    *pi;

pi = new int[5] = {1, 2, 3, 4, 5};
```

3. 当内存分配成功后，一般情况下不允许改变指针 pointer 的值，除非在程序中提供了恢复其初值的的手段。这是因为对修改过的指针实施 delete 运算时，将会引起系统内存管理的混乱。这一点在使用时应特别注意。比如：

```
int    *p = new[10], temp;

temp = p;                    // 保存指针 p 的值
p += 3;

delete temp;                 // 返还动态内存
```

若没有指针 temp，则最后一条语句就只能是 delete p。但由于指针 p 的值已经改变，不再指向 new 运算符所分配内存的首地址，其后果将是不可预知的。运算符 delete 不像 new 那样返回一个值来表示运算的结果，它不返回任何值。因此，正确地使用 new 和 delete 则是程序员的责任。

例 6.20 根据用户输入建立一个学生成绩单。

```
//      EX6_20.CPP

#include      <iostream.h>

void   main()
{
       int     n, i, j;
       float   (*p)[6], *q;

       cout << "Enter number of student: ";
       cin   >> n;

       p = new float[n][6];
       for(i = 0; i < n; i ++)  {
              cout << "Enter student #" <<(i + 1) << " scores:\n";
              q = p[i];
              q[4] = 0;
              for(j = 0; j < 4; j ++)  {
                     cout << "Score No. " << (j + 1) << ": ";
                     cin >> q[j];
                     q[4] += q[j];
```

```
            }
            q[5] = q[4] / 4;
        }

        cout << "No.\tSc1\tSc2\tS3\tS4\tTotal\tAverage\n\n";
        for(i = 0; i < n; i ++)  {
            cout <<(i + 1);
            q = p[i];
            for(j = 0; j < 6; j ++)
                cout << '\t' << q[j];
            cout << endl;
        }

        delete p;
}
```

该程序根据用户所输入的 n 值，利用 new 运算符动态地生成一个 n×6 的二维数组，其中，n 为学生的人数，第二维中的前 5 个元素用来存放一个学生的 5 门功课的成绩，最后一个元素用来存放这 5 门功课的平均值。

应当说明的是：在C语言中也有一些与运算符 new 和 delete 功能相似的库函数，为了保持兼容性，C++语言也保留了这些函数并且允许在C++程序中调用它们。比如，C 语言的库函数 malloc（）的作用就与 new 相同，而库函数 free（）的作用就与 delete 相同。但是，由于它们对内存管理的策略不同，因此两者的混用将会导致系统的崩溃。比如，将由 new 动态分配的内存用函数 free（）返还给系统，或者将调用函数 malloc（）申请到的内存用 delete 释放，都将导致一个严重的错误。

6.5 引用

C++语言提供了一个与指针密切相关的特性——引用。在计算机术语中，引用指的是对一个名子的访问，而这里所说的引用则相当一个变量。它提供了一种访问数据的手段，特别是为函数间传递数据提供了方便。

6.5.1 引用的说明与使用

引用运算符“&”用来说明一个引用。除了在函数说明中的函数值和参数外，引用的说明必须是定义性说明，即必须对它进行初始化。例如：

```
int    i, &refi = i;
```

这里，引用 refi 被说明为对变量 i 的引用，其引用类型为 int 型。经过这样的说明以后，变量 i 与引用 refi 代表的是同一个变量。实际上，refi 只不过是变量 i 的一个别名而已。

例6.21 使用引用。

```
//      EX6_21.CPP

#include      <iostream.h>
```

```
void   main()
{
       int     i = 100, &refi = i;

       cout << refi << endl;
       refi += 100;
       cout << i;
}
```

该程序的输出为：

```
100
200
```

由上述程序的输出可见，访问引用与访问它所引用的变量一样。实际上，引用不是一个变量，它本身是不能被修改的，对一个引用的访问实质上就是对它所引用的变量的访问，这就是引用这一叫法的来由。作为对照，让我们再来看一个例子。

例 6.22 例 6.21 的另一版本。

```
//      EX6_22.CPP

#include      <iostream.h>

void   main()
{
       int     i = 100, *refi = &i;

       cout << *refi << endl;
       *refi += 100;
       cout << i;
}
```

例 6.22 与例 6.21 的功能完全相同，只不过是将引用换成了指针。比较这两个例子我们可以发现，引用和指针的性质非常相似。实际上，编译器在处理引用与处理指针时的方法也基本相同。但在源程序中，使用引用就显得比使用指针来得自然。

使用引用应注意：

1. 由于引用不是一个变量，所以不以能说明对引用的引用、元素为引用的数组和指向引用的指针。比如，以下的说明都是错误的：

```
int    &&ref;              // 企图说明一个引用的引用
int    &refa[5];           // 企图说明一个引用数组
int    &*refp;             // 企图说明一个指向引用的指针
```

但是，可以说明对指针的引用：

```
int   *pa, *&refp = pa;
```

2. 一般讲，为引用指定的初值必须是一个与其引用类型相同的变量。若将一个常量或是一个常量标识符指定为一个引用的初值，或者将一个不同类型的变量指定为一个引用的初

值，则编译器将会建立一个临时变量，然后将所指定的值赋给该临时变量。引用则是引用该临时变量，对引用的访问就是对该临时变量的访问。注意：对此编译器会发出一个警告信息。

3. 可以利用动态内存分配来初始化一个引用，这时就可以将引用看作是动态建立起来的变量的名子。

例 6.23 引用的特殊使用。

```
//      EX6_23.CPP

#include      <iostream.h>

void   main()
{
       const int       ci = 100;
       float   f = 55.5;
       int     &ri = ci, &rf = f;

       ri += 100;
       rf += 40
       cout << "CI = " << ci << "\tRI = " << ri << endl;
       cout << "F = " << f << "\tRF = " << rf << endl;

       float   &reff = *new float(66.6);                    // A

       cout << "REFF = " << reff << endl;

       delete &reff;                                        // B
}
```

该程序的输出为：

```
CI = 100    RI = 200
F = 55.5    RF = 95
REFF = 66.599998
```

输出中最后一行中的值是由于实数的存储误差引起的。

程序中行 A 就是利用动态内存分配来初始化引用 reff 的。注意：这时运算符 new 前边的星号是必须的。这是因为 new 返回的是一个指针，而引用 reff 则引用的是一个实型变量，这里星号是表示用指针的内容来初始化引用。同样的道理，当 delete 一个由 new 初始化的引用时，必须在引用前边加取地址运算符“&”，如程序中行 B 所示。

6.5.2 引用与函数

C++语言引入引用这一特性的主要目地是为了为函数间传递数据提供方便。在应用中也主要是用作函数的值和函数的参数。

6.5.2.1 引用作为函数的参数

函数的参数可以说明为引用类型，引用参数实参与形参的结合属于引用调用。引用调用与地址调用相似，函数中对形参的修改将直接影响实参的值。然而与地址调用相比，引用调用显得更加方便、自然。

例 6.24 例 6.9 的另一版本。

```
//      EX6_24.CPP

#include      <iostream.h>

void   main()
{
       void   Swap(int&, int&);

       int    a = 3, b = 5;

       cout << "Before SWAP be called:\ta = " << a << "\tb = " << b << endl;
       Swap(a, b);                                                    // A
       cout << "After SWAP be called:\ta = " << a << "\tb = " << b << endl;
}

void   Swap(int &x, int &y)
{
       int    temp = x;

       x = y;
       y = temp;
}
```

程序 EX6_24 的作用与例 6.9 中程序的作用完全相同，不同的是前者中函数 Swap（）的参数为引用，而后者则为指针。比较两个程序中 Swap（）函数的定义、行 A 中对该函数的调用，可以看出对于同一目地——在函数中改变实参的值，使用引用要比使用指针来得方便、直观。

6.5.2.2 值为引用的函数

将引用作为函数的返回值最能体现引用的特性。无论函数的值是一个数还是一个地址，从本质上来讲都是一个数值。若函数的返回值为一个引用，则可以理解为函数返回了一个变量。因此，值为引用的函数可以用作赋值运算符的左操作数。

例 6.25 值为引用的函数。

```
//      EX6_25.CPP

#include      <iostream.h>

void   main()
{
       int    &func(void);

       int    i;

       for(i = 0; i < 5; i ++)
              cout << func() << '\t';
       cout << endl;
```

```
    func() = 100;                          // A

    for(i = 0; i < 5; i ++)
        cout << func() << '\t';
    cout << endl;
}

int    &func(void)
{
    static int      count = 0;

    return ++ count;                       // B
}
```

该程序的输出为：

```
1    2    3    4    5
101  102  103  104  105
```

在例 6.25 中，函数 func（）是一个值为引用的函数。程序中行 B 的 return 语句不是返回变量 count 的值，而是返回对它的引用，即返回变量 count 本身。因此程序中行 A 实际上就是向函数 func（）中的局部变量 count 赋值 100，从而使得程序中第二个循环所输出的值如输出中第 2 行所示。

注意：函数不应该返回对局部自动变量的引用，其原因与函数不应该返回指向局部自动变量的指针一样。

6.6 void 和 const 型指针

一个指针可以说明成 void 或 const 的，前者称作 void 型指针，后者称作 const 型指针。

6.6.1 void 型指针

void 型指针也叫作无类型指针，它可以接受任何类型指针的值。但若将 void 型指针的值赋给其他类型的指针时，则必须进行强制类型转换。例如：

```
int    *pi;
void   *pv;

pv = pi;                   // 正确
pi = pv;                   // 错误
pi = (int *)pv;            // 正确
```

C++提供 void 型指针是为了便于编写通用函数。先看一个例子：

例 6.26 使用 void 型指针编写通用函数。

```
//      EX6_26.CPP

#include      <iostream.h>
```

```
void   main()
{
        void   Sort(void*, unsigned, int, int(*)(void*, void*));
        int    Cmp_Int(void*, void*);
        int    Cmp_Float(void*, void*);

        int    ia[] = {23, 54, 38, 89, 66, 32}, i;
        float  fa[] = {34.5, 33.8, 12.3, 89.6, 72.1, 15.4, 38.6, 44.5};

        for(i = 0; i < sizeof(ia) / sizeof(int); i ++)
                cout << ia[i] << '\t';
        cout << endl;

        Sort(ia, sizeof(ia) / sizeof(int), sizeof(int), Cmp_Int);

        for(i = 0; i < sizeof(ia) / sizeof(int); i ++)
                cout << ia[i] << '\t';
        cout << endl;

        for(i = 0; i < sizeof(fa) / sizeof(float); i ++)
                cout << fa[i] << '\t';
        cout << endl;

        Sort(fa, sizeof(fa) / sizeof(float), sizeof(float), Cmp_Float);

        for(i = 0; i < sizeof(fa) / sizeof(float); i ++)
                cout << fa[i] << '\t';
        cout << endl;
}

void   Sort(void *arr, unsigned n, int siz, int (*comp)(void*, void*))
{
        int    i, j, k;
        char           *p, *q, temp;

        for(i = 0; i < n - 1; i ++)       {
                p = (char*)arr + i * siz;                              // A
                for(j = i + 1; j < n; j ++)       {
                        q = (char*)arr + j * siz;                      // B
                        if(comp(p, q) > 0)
                                for(k = 0; k < siz; k ++)       {      // C
                                        temp = p[k];
                                        p[k] = q[k];
                                        q[k] = temp;
                                }
                }
        }
}

int    Cmp_Int(void *a, void *b)
```

```
{
    return *(int*)a - *(int*)b;
}

int    Cmp_Float(void *a, void *b)
{
    return (*(float*)a - *(float*)b < 0 ? -1 : 1);
}
```

该程序的输出为：

```
23    54    38    89    66    32
23    32    38    54    66    89
34.5  33.799999  12.3  89.599998  72.099998  15.4  38.599998  44.5
12.3  15.4  33.799999  34.5  38.599998  44.5  72.099998  89.599998
```

程序中的 Sort（）函数是一个通用的对数组进行冒泡排序的函数。关于冒泡排序法的原理已超出了本书讨论的范围，有兴趣的读者可以参阅有关“数据结构”的资料。由于 Sort（）函数事先并不知道所需排序之数组的类型，所以它将这组数据视为 void*型的。参数 arr 是指向这一数组首地址的指针；n 为数组中元素的个数，一个类型为 type 的数组 array 的大小可以通过 sizeof(array) / sizeof(type) 计算出来；siz 是数组元素以字节计的大小，其值可以通过 sizeof(type)计算出来。行 A 是将数组中下标为 i 之元素的首地址赋给字符型指针 p；行 B 则是将数组中下标为 j 之元素的首地址赋给字符型指针 q。只所以将指针 p 和 q 说明字符型的，是因为 char 型数据在计算机中占据一个字节的内存单元，是计算机中存取数据的最小单位，所有类型数据的大小都必定是一个 char 型数据的整倍数。在排序过程中，若数据在数组中的位置不正确，则需要交换数据的位置，函数中的循环 C 就是用来交换两个数据的位置的。由于事先并不知道数据的具体类型，所以交换也是按字节进行的。

函数 Cmp_Int（）和 Cmp_Float（）是分别用来比较两个整型数据和两个实型数据的大小的。若在一般情况下，它们的参数可以分别说明成整型的和实型的变量、指针或是引用。但是由于它们是要供函数 Sort（）来使用的，而 Sort（）函数事先并不知道将要排序的数组是什么数据类型的，它的参数 comp——函数指针——中的参数就只能说明成 void*的。因此，函数 Cmp_Int（）和 Cmp_Float（）中的参数为了与 Sort（） 函数中的参数相匹配，也必须说明成 void*的，而在函数体中再将它们强制转换成相应的类型。同样，为适应函数 Sort（），这两个函数的返回值也必须说明成同一类型——int 型，并规定函数值大于零、等于零和小于零分别表示第一个参数的值大于、等于和小于第二个参数的值。读者可能会问：函数 Cmp_Float（） 的函数体为什么不能像函数 Cmp_Int（）的那样，却要搞得这么复杂呢？这是因为该函数的值为 int 型，而它的两个参数之差为一 float 型，若将这一差值以一个 int 型的形式返回，由于存在类型转换时的误差，就有可能造成判断的失误。例如 34.5 - 33.8 的值为 0.3，将此值转换成 int 型时则为 0，于是在函数 Sort（）中因比较函数的值不大于 0 而认为前者不大于后者，因而也就不交换它们的位置，最终导致排序的结果是错误的。

由程序 EX6_26 的输出可以看出：由于采用了 void 型指针，使得函数 Sort（）既可以对整型数组排序，又可以对实型数组排序。实际上，该函数可以对任何类型的数组进行排序。作为一个练习，请读者利用该函数对字符型数组和字符串数组进行排序，其中的比较函数由

读者自己编写。

6.6.2 const 型指针

用关键字 const 修饰一个指针时，其位置的不同将表示不同的意义。比如：

```
const char   *pc = "Point to a constant;
char  *const       pc1 = "A constant pointer";
const char   *const       pc2 = "A constant pointer points to a constant";
```

指针 pc 被说明成一个指向常量的字符型指针。即它所指的内容为常量，是不可修改的。因此，如下的赋值属于非法操作：

```
*pc = 'a';          // 非法
```

但 pc 本身却是一个变量，对它的值的修改则是合法的：

```
pc = qc;            // 合法
```

这里，qc 也须是一个指向常量的字符型指针，否则将属于非法操作。然而，利用强制类型转换则可以将任何类型的指针赋给 pc：

```
pc = (const char*)AnyPointer;
```

其中：AnyPoint 是一个任意类型的指针。

指针 pc1 被说明成一个字符型的常指针，即它的值是一个常量，永远是一个固定的内存地址。因此，如下的赋值将是非法操作：

```
pc1 = qc1;          // 非法
```

无论 qc1 与 pc1 的类型是否相同。然而，pc1 所指的内容却是一个变量，修改其内容的操作则是合法的：

```
*pc1 = 'a';         // 合法
```

由于常指针本身不允许修改，所以在说明时必须对它进行初始化，除非它是一个函数的形参。

指针 pc2 被说明成一个指向常量的常指针，因此，无论是它的值还是它的内容均不允许修改。

说明 const 型指针的目的是为了防止在程序中无意识地改变了它的值或它的内容，以提高数据的安全性。同时，编译器也可以利用该信息产生更有效的代码。

以上的内容同样适用于引用。使用 const 型引用在有些场合下还可以提高数据传递的效率。例如：

```
void  f(const long &l);
```

就将函数 f（）的参数说明成长整型的常引用，因而可以有效地阻止在函数中对参数 l 的无意修改。当然，采用传值的方式也可以起到相似的作用，但采用引用的效率更高。因为，对于 long 型数据，传值时需要传递 4 个字节的数据；而引用调用时，由于实际上传递的是实

参的地址，因而只需要传递两个字节的数据即可。对于 double 型和 long double 型的数据，引用调用的效率将更加明显。

习题

6-1 设有说明：

```
int    i = 50, *ip = &i;
```

并设变量 i 的在内存中的起始地址为 2500，则 ip 和*ip 的值各为多少？

6-2 设有说明：

```
int    i[3] = {50, 60, 70}, *ip = i;
```

则经过以下的赋值后，j 的值分别为多少？

```
j = *ip ++;          j = (*ip)++;          j = *(ip++);
```

6-3 将第 5 章习题 5-2～5-9 中函数的参数改为指针。

6-4 编写一个函数

```
char* isIn(char *str, char ch);
```

该函数测试字符串 str 中是否存在参数 ch 所指定的字符。若存在，则返回指定字符在串中第一次出现的位置；否则返回空指针。

6-5 编写一个函数

```
char* isSubStr(char *str1, char *str2);
```

该函数测试字符串 str2 是否为字符串 str1 的子串。若是，则返回 str2 在 str1 中的首地址；否则返回空指针。

6-6 说明一个可以存放 10 个字符串的字符串数组，各字符串由用户输入，数组中每个元素的大小根据用户输入串的实际长度动态地确定。（提示：利用动态内存分配）。

6-7 编写一个函数：

```
void  SortStr(char **sArr, int n);
```

该函数对习题 6-6 中的字符串数组进行排序，参数 n 为数组元素的个数。（提示：排序时无须交换两个串，只需交换指向它们的指针即可）。

6-8 编写 4 个分别带有两个整型参数并返回整型值的函数，其返回值分别为两个参数的和、差、积、商；说明一个包含 4 个元素的函数指针数组，各元素分别指向上述 4 个函数。编写程序，要求用户输入形如：

```
12 + 34
```

的表达式，输出表达式的运算结果。

6-9 利用动态内存分配技术改写第 5 章习题 5-2，要求能处理任意人数的学生成绩，人数由用户输入。

6-10 利用动态内存分配技术改定第 5 章习题 5-3，要求能处理任意人数及任意门课程的成

绩，人数与课程门数由用户输入。（提示：使用指向指针的指针）。

6-11 试述指针与引用的同异。

6-12 将习题 6-4 所要求的函数改写成：

```
char&       isIn(char *str, char ch);
```

编写程序，将字符串 "dwajsdfadjfartuare" 中所有的字符 'a' 改为 'b'。

6-13 编写一个程序，统计一篇英文文章中的行数的单词数。

6-14 试为例 6.26 增加对字符型数组、字符串数组、long 型数组和 double 型数组的排序功能。

6-15 利用函数重载改写例 6.26 及习题 6-14。

第 7 章　结构、联合和枚举

C++语言的导出数据类型除了数组、指针外，还有结构、联合、枚举和类等，另外，还提供了由用户自行定义数据类型的手段。这一切为程序设计中构造复杂数据结构提供了极大的方便。本章介绍这方面的内容。

为了后边论述的方便，本章首先介绍类型定义。

7.1　类型定义

关键字 typedef 用于定义一个标识符来代表一种数据类型，经它定义的标识符可以像 C++语言的数据类型名一样用在以后的变量说明和类型定义中。这就是所谓的类型定义。

用 typedef 定义一个标识符的一般形式为

typedef　　type　identifier;

其中：type 是任一合法的 C++数据类型——基本数据类型、导出数据类型以及经 typedef 定义的标识符；identifer 为一个标识符。例如：

```
typedef     int     integer;
```

就将标识符 integer 定义为数据类型 int。

经过这样的定义，变量说明：

```
integer     i;
```

就与说明：

```
int    i;
```

完全等价，即把标识符 i 说明成一个整型变量。

使用类型定义主要是为了改善程序的可读性和简化程序的书写，它并没有为 C++语言产生任何新的数据类型，定义过的标识符只不过是已有数据类型的一个别名而已。比如，integer 就比 int 显得直观。再如，经定义：

```
typedef     unsigned long     UL;
```

则变量说明：

```
UL    l;
```

就要比：

```
unsigned long     l;
```

便于书写。

类型定义用于定义一个函数指针将显得更有效率。例如，在例 6.26 中可以这样定义：

```
typedef    int    (*cmp)(int*, int*);
```

即将标识符 cmp 定义成一个指向带有两个整型指针参数并返回整型值之函数的函数指针。该定义等价于：

```
typedef    int    (*)(int*, int*)      cmp;
```

这样，该例中函数 Sort（）的函数原型就可以写为

```
void  sort(void*, unsigned, int, cmp);
```

而该函数定义时的函数头可以改写成

```
void  sort(void *vect, unsigned n, int siz, cmp comp)
```

typedef 还可以同时对一个类型给出多个定义。比如：

```
typedef    int    Integer, INT;
```

就与：

```
typedeg    int    Integer;
typedef    int    INT;
```

两条语句完全等效。更为方便的是，在一条 typedef 语句中还可以同时定义同一类型的指针和引用。比如

```
typedef    int    Integer, *Pint, &Rint;
```

就等效于：

```
typedef    int    Integer;
typedeg    int*   Pint;
typedef    int&   Rint;
```

类型定义很像宏定义，编译器对类型定义的处理也与宏定义相似，即将所有经类型定义的标识符先进行转换再进行编译。但两者还是有一些区别的：首先，类型定义是C++语言中的语句，它可以出现在程序中任何语句可以出现的地方，而宏定义则属于编译预处理指令，它必须独占一行；其次，可以定义带参数的宏，但没有带参数的类型定义。

7.2 结构

数组是一个同类型数据的集合，但在一些应用中，常需要将一些不同类型的数据组合成一个有机的整体。比如，数据库中关于一个学生的纪录，它应该包括学生的姓名、性别、年龄、成绩等数据项，这些数据项分别属于不同的数据类型：姓名就是一个字符串，性别应该是一个字符，而年龄和成绩则分别应当是整型和实型数据。C++语言所提供的结构类型就可以满足这一要求。

7.2.1 定义结构

与数组不同，一个结构必须先经定义后才可以说明属于这种结构类型的变量，这是由于结构的组成是千变万化的。

关键字 struct 是用来定义一个结构的。定义一个结构具有以下的一般形式：

```
struct      struct_name      {
      members;
};
```

其中：struct_name 是程序员为所定义的结构起的名字，它应符合 C++语言对标识符的要求；members 是结构中的一个个数据项，它们都是一些变量，叫作结构的成员。例如：

```
struct      student     {
      char  name[20];
      char  sex;
      int   age;
      float score;
};
```

就定义了一个名为 student 的结构。该结构有 4 个成员：字符串 name 用来保存姓名；字符型变量 sex 用来存放性别（可用 m 表示男性，f 表示女性）；整型变量 age 用来存放年龄；而实型变量 score 则用来保存成绩。

结构的定义明确地描述了该结构的组织形式，它告诉编译器属于这种组织形式的变量使用内存的模式。比如，结构 student 的定义就通知编译器属于这种类型的变量在内存中的存储形式如图 7-1 所示。

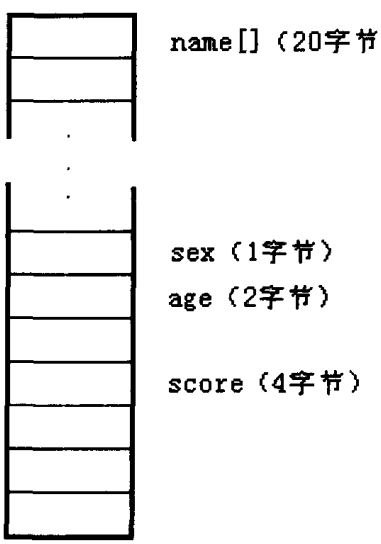

图 7-1 结构 student 的存储形式

由图 7-1 可见，一个 student 型结构变量一共需要 27 个字节的存储单元，这个结论也可以通过运算 sizeof（student）得出。

在一个程序中，一旦定义了一个结构，就好像为语言增加了一个新的数据类型，程序中就可以说明属于这种数据类型的变量。

7.2.2 说明结构变量

定义一个结构仅仅是指定了结构的组织形式，它并不占据内存，因此在程序中也就无法对它进行访问。程序可访问的是结构变量，结构变量与其他变量一样，必须先行说明。说明一个结构变量的一般形式为

```
<storage> struct_name      var_name<, var_name...>;
```

比如，设程序中已经定义过了结构 student，于是就可以说明 student 型的结构变量：

```
student    stu1, stu2;
```

这里 stu1 和 stu2 就是结构变量，编译器将为它们分配内存。

也可以在定义结构的同时来说明结构变量。比如：

```
struct       student      {
        char   name[20];
        char   sex;
        int    age;
        float  score;
}       stu1, stu2;
```

若在定义结构时就说明变量，还允许省去结构名：

```
struct {
        char   name[20];
        char   sex;
        int    age;
        float  score;
}       stu1, stu2;
```

但这种说明由于没有定义结构名，所以在程序的其他部分就再也不能说明类型属于这种结构类型的结构变量了，因此，这种说明通常只用来说明具有块作用域的结构及其变量。

在说明一个结构变量的同时也可以对它进行初始化，例如：

```
student      stu = {"Wang Ping", 'm', 22, 87.5};
```

就说明了一个 student 型结构变量 stu，并将它的 name 成员的值初始化为字符串"Wang Ping"；sex 成员的的值初始化为字符'm'；age 成员的值初始化为 22；score 成员值初始化为 87.5。

结构变量的初始化同样适合其他两种结构变量说明形式。

7.2.3 访问结构变量

访问一个结构变量包含两个方面的操作：访问整个结构变量和访问一个结构变量中的某个成员。

访问整个结构变量通常用于结构变量之间的赋值和在函数之间传递结构变量。一个结构变量可以将它所有成员的值一次性地赋给另一个类型相同的结构变量。比如：

```
stu1 = stu;
```

就将变量 stu 的全部成员的值赋给了变量 stu1。这里 stu1 也是一个 student 型的结构变量。

关于在函数间传递结构变量将在 7.3 节中介绍。

访问一个结构变量中的某个成员需要用到成员选择运算符“.”，它的使用具有如下的一般形式：

var_name.member

比如：stu.name，stu.sex，stu.age 和 stu.score 就是分别访问结构变量 stu 中的每一个成员的值。

例 7.1 对结构变量的访问。

```
//      EX7_1.CPP

#include      <iostream.h>

void   main()
{
       struct  student        {
              char    name[20];
              char    sex;
              int     age;
              float   score;
       }      stu1 = {"Wang Ping", 'm', 22, 87.5};

       student        stu2 = {"Liu Bing", 'f', 21, 89}, stu3;
       char    *name = "Zhang Gaofeng";
       int     i;

       for(i = 0; name[i] != '\0'; i ++)                    // A
              stu3.name[i] = name[i];
       stu3.name[i] = '\0';
       stu3.sex = 'm';
       stu3.age = 23;
       stu3.score = 92.5;

       cout << "STU1\t" << stu1.name << '\t' << stu1.sex << '\t' << stu1.age << '\t' << stu1.score << endl;
       cout << "STU2\t" << stu2.name << '\t' << stu2.sex << '\t' << stu2.age << '\t' << stu2.score << endl;
       cout << "STU3\t" << stu3.name << '\t' << stu3.sex << '\t' << stu3.age << '\t' << stu3.score << endl;

       stu3 = stu1;                                         // B
       cout << "\nSTU3\t" << stu3.name << '\t' << stu3.sex << '\t' << stu3.age << '\t' << stu3.score << endl;
}
```

该程序的输出为：

```
STU1:     Wang Ping       m     22    87.5
STU2:     Liu Bing        f     21    89
STU3:     Zhang Gaofeng   m     23    92.5

STU3:     Wang Ping       m     22    87.5
```

需要指出的是：C++语言中预定义的输出流 cout 由于不可能预先知道用户所定义之结构的具体形式，所以它不可能对一个结构变量进行整体输出（除非进行了运算符重载。这将在第 12 章中介绍），只能逐一地输出结构变量中的各个成员的值。

程序中的循环 A 是用来为结构变量 stu3 的成员 name 赋值的，由于该成员是一个数组变量，不能对它进行整体赋值，所以只能用一个循环逐个元素地为它赋值。但从行 B 的赋值操作可以看出：无论一个结构中是否包含有数组成员，结构变量间的赋值操作总是可以整体进行的。

7.2.4 结构变量成员

结构的成员可以是任何合法的 C++变量，结构变量是 C++的变量，因此，结构变量也可以作为一个结构的成员。

用一个结构的结构变量作为另一个结构的成员可以构造更为复杂的数据结构。比如，设某通讯录具有如图 7.2 所示的组织形式：

姓名	工作单位			家庭住址		
	邮编	地址	电话	邮编	地址	电话

图 7-2 某通讯录的组织形式

则该通讯录可以采用如下的结构来表示：

```
struct      p_a_t {
        unsigned long     zip;        // 邮编
        char  *addr;                  // 地址
        unsigned long     tel;        // 电话
};

struct      address_book     {
        char  *name;                  // 姓名
        p_a_t depart;                 // 工作单位
        p_a_t home;                   // 家庭住址
};
```

这里，表示通讯录的结构 address_book 中，代表工作单位和家庭住址的成员 depart 和 home 就又是类型为 p_a_t 的结构变量。这种用结构变量作为结构成员的方式又叫作结构的嵌套。

访问一个嵌套结构中的成员时，需要一层层地访问。比如，设变量 a_b 是一个 address_book 类型的结构变量，则访问它的 name 成员的形式为

```
a_b.name
```

而访问它的 depart 中的 tel 成员的形式则为

```
a_b.depart.tel
```

若某结构包含有多层嵌套，则对它的成员访问也是按多层地进行。

例 7.2 使用嵌套的结构。

```
//      EX7_2.CPP

#include      <iostream.h>
#include      <string.h>

struct  p_a_t  {
        unsigned long          zip;
```

```
        char    *addr;
        unsigned long           tel;
};

struct address_book {
        char    *name;
        p_a_t   depart;
        p_a_t   home;
};

void    main()
{
        address_book            a_b;
        char    str[81];
        int     len;

        cout << "Please input name: ";
        cin >> str;                                     // A
        len = strlen(str);                              // B
        a_b.name = new char[len + 1];                   // C
        strcpy(a_b.name, str);                          // D

        cout << "Plase input depart information:\n";
        cout << "Post code: ";
        cin >> a_b.depart.zip;
        cout << "Address: ";
        cin >> str;
        len = strlen(str);
        a_b.depart.addr = new char[len + 1];
        strcpy(a_b.depart.addr, str);
        cout << "Telephone: ";
        cin >> a_b.depart.tel;

        cout << "Plase input home information:\n";
        cout << "Post code: ";
        cin >> a_b.home.zip;
        cout << "Address: ";
        cin >> str;
        len = strlen(str);
        a_b.home.addr = new char[len + 1];
        strcpy(a_b.home.name, str);
        cout << "Telephone: ";
        cin >> a_b.home.tel;

        cout << "NAME: " << a_b.name << endl;
        cout << "DEPARTMENT:\n";
        cout << "\tPost Code: " << a_b.depart.zip << "\tAddress: " << a_b.depart.addr
                << "\tPhone Number: " << a_b.depart.tel << endl;
        cout << "HOME:\n";
        cout << "\tPost Code: " << a_b.home.zip << "\tAddress: " << a_b.home.addr
                << "\tPhone Number: " << a_b.home.tel << endl;
```

```
    delete a_b.name;
    delete a_b.depart.addr;
    delete a_b.home.addr;
}
```

本例中，为了节省存储空间，将 p_a_t 结构中的 addr 成员和 address_book 结构中的 name 成员说明成字符型指针，这就使得每当将一个表示姓名或地址的字符串赋给它们之前，先得为它们动态地分配一块内存，然后将字符串复制到该内存中。例如对 name 成员的赋值：程序中的行 A 是先将用户输入的字符串存放在一个字符型数组 str 中；行 B 通过调用库函数 strlen（）求得该串的长度，并将函数的返回值赋给变量 len；行 C 为 name 成员分配一块大小为 len + 1 个字节的内存单元，多出的一个字节是为了能存放字符串结束标志；行 D 则是通过调用库函数 strcpy（）将 str 中的内容复制到 name 所指的内存块中。

这种利用字符型指针来代替字符型数组的方法可以最大限度地节省内存，因为实际占有的内存量与字符串所需的内存量完全相等，若使用字符型数组则可能有些元素所占的内存单元被白白地浪费掉了。更为严重的是，若字符型数组预先定义的过小，还可能导致下标超界等错误。对于像结构、类这样的复杂数据结构，利用动态内存分配技术应当是一个最佳选择。应当说明的是，在例 7.2 中为了节省篇幅，对 new 运算符的执行成功与否没有进行检查。在一个实用的程序中必须进行这样的检查，以保证程序的可靠性。利用 new 运算符动态分配到的内存必须利用 delete 运算符将它们返回还给操作系统，EX7_2.CPP 中最后 3 条语句就是为此目的而设置的。

7.3 结构数组和结构指针

一个结构就如同 C++语言的一个数据类型，因此可以说明元素为结构的数组和指向结构的指针。

7.3.1 结构数组

元素为结构的数组称为结构数组。说明一个结构数组的一般形式为

<storage> struct_name array_name[constant1][constant2]...[constantn];

例如：

```
student     sts[30];
```

就说明了一个包含有 30 个元素的结构数组，该数组可以用来记录一个班学生的情况。

在说明结构数组的同时也可以顷它进行初始化：

```
student     sts[3] = {{"Wang Ping", 'm', 22, 87.5},
                      {"Liu Bing", 'f', 21, 89},
                      {"Zhang Gaofeng", 'm', 23, 92.5}};
```

或者：

```
student     sts[] = {"Wang Ping", 'm', 22, 87.5,
```

```
                "Liu Bing", 'f', 21, 89,
                "Zhang Gaofeng", 'm', 23, 92.5};
```

访问一个结构数组的方法综合了数组元素的访问方式和结构变量的访问方式，比如，访问数组 sts 中下标为 5 之元素的成员 name：

```
sts[5].name
```

例 7.3 建立一个学生情况表。

```
//      EX7_3.H

#if !defined   _EX7_3_H_
#define        _EX7_3_H_

struct  student
        char    *name;
        char    sex;
        int     age;
        float   score;
};

#endif


//      EX7_3.CPP

#include       <iostream.h>
#include       <string.h>
#include       "ex7_3.h"

void    main()
{
        studentsts[30];
        char    str[81];
        int     i;

        cout << "Please enter datas, separate with space\n";
        cout << "\tName\tSex Age Score\n";
        for(i = 0; i < 30; i ++) {
                cout << "No." << (i + 1) << ":\t";
                cin >> str >> sts[i].sex >> sts[i].age >> sts[i].score;
                sts[i].name = new char[strlen(str) + 1];
                strcpy(sts[i].name, str);
        }
        cout << endl;

        for(i = 0; i < 30; i ++)
                cout << (i + 1) << '\t' << sts[i].name << '\t' << sts[i].sex
                        << '\t' << sts[i].age << '\t' << sts[i].score << endl;
```

```
    for(i = 0; i < 30; i ++)
        delete sts[i].name;
}
```

该程序要求用户逐一输入每个学生的姓名、性别、年龄和成绩，然后再将它们一一输出。注意：运行该程序输入学生姓名时，不得输入包含有空格的字符串，就是用双引号将它们括起来都不行。应利用连字符或下划线来代替空格，例如：

Wang_Ping

7.3.2 结构指针

类型为结构的指针叫作结构指针。说明一个结构指针具有以下的一般形式：

<storage> struct_name *pointer;

例如：

```
student     *sp;
```

就说明了一个类型为 student 的结构指针 sp。

与普通指针一样，以写方式访问一个结构指针时，该指针必须具有明确的值。通常，在说明一个结构指针的同时可以为它赋一初值：

```
student    stu, *sp = &stu;
```

可以采用以下方式访问一个结构指针的内容：

```
(*sp).name
```

表示对结构成员 stu.name 的访问。注意：由于递引用运算符“* ”的优先级低于成员选择运算符“.”，所以表达式中的圆括号是必不可少的，否则编译器将把它解释成访问结构变量 stu 成员 name 中的第一个字符。

访问结构指针更常用的是 C++语言所提供的另一个成员选择运算符——“->”，该运算符是专门用于结构指针和类指针的。例如，上述访问可以改写为

```
sp->name
```

例 7.4 例 7.3 的另一版本。

```
//      EX7_4.CPP

#include      <iostream.h>
#include      <string.h>
#include      "ex7_3.h"

void    main()
{
        student         sts[30], *sp = sts;
        char    str[81];
        int     i;
```

```
    cout << "Please enter datas, separate with space\n";
    cout << "\tName\tSex Age Score\n";
    for(i = 1; i <= 30; i ++)        {
        cout << "No." << i << ":\t";
        cin >> str >> sp->sex >> sp->age >> sp->score;
        sp->name = new char[strlen(str) + 1];
        strcpy(sp->name, str);

        sp ++;
    }
    cout << endl;

    for(i = 1, sp = sts; i <= 30; i ++, sp ++)
        cout << i << '\t' << sp->name << '\t' << sp->sex
            << '\t' << sp->age << '\t' << sp->score << endl;

    for(i = 0; i < 30; i ++)
        delete sts[i].name;
}
```

结构指针的一个典型应用就是将一个链表或树的结点定义成一个结构，该结构中有一个或多个结构指针用作结点的链接域。关于这方面的具体应用将在下一节举例介绍。

7.4 结构与函数

结构变量可以用作函数的参数，函数也可以返回一个结构的值。函数中结构参数的实参与形参的结合方式属于赋值调用。

例 7.5 例 7.3 的又一版本。

```
//      EX7_5.H

#if !defined    _EX7_5_H_
#define         _EX7_5_H_
#include        "ex7_3.h"

student         Input_Data(void);
void    Output_Data(student);
void    Delete_Data(student);

#endif


//      EX7_5A.CPP

#include        <iostream.h>
#include        <string.h>
#include        "ex7_3.h"

student         Input_Data(void)
```

```
{
        static student  st;
        char    str[81];

        cin >> str >> st.sex >> st.age >> st.score;
        st.name = new char[strlen(str) + 1];
        strcpy(st.name, str);

        return st;
}

void    Output_Data(student st)
{
        cout << st.name << '\t' << st.sex << '\t' << st.age << '\t' << st.score << endl;
}

void    Delete_Data(student st)
{
        delete st.name;
}


//      EX7_5.CPP

#include      <iostream.h>
#include      "ex7_5.h"

void    main()
{
        student         sts[30];
        int     i;

        cout << "Please enter datas, separate with space\n";
        cout << "\tName\tSex Age Score\n";
        for(i = 0; i < 30; i ++) {
                cout << "No." << (i + 1 )<< ":\t";
                sts[i] = Input_Data();                          // A
        }
        cout << endl;

        for(i = 0; i < 30; i ++) {
                cout << (i + 1) << '\t';
                Output_Data(sts[i]);
        }

        for(i = 0; i < 30; i ++)
                Delete_Data(sts[i]);
}
```

程序中当用实参 sts[i]调用函数 Output_Data（）时，系统就将 sts[i]的所有成员逐一赋给形参 st 的对应成员，相当于存在一个 st = sts[i] 的操作。在行 A 中调用函数 Input_Data（）

时，由于该函数返回其局部静态变量 st 的值，因此就相当于存在一个 sts[i] = st 的操作。注意：文件 EX7_5A.CPP 中在定义函数 Input_Data（）时将结构变量 st 说明成静态的，这里必须这样做。其原因请读者处行分析。

当一个结构中包含有大量的数据时，结构作为函数的参数或函数的值由于存在有众多的赋值操作从而影响程序运行的效率，这时可以采用引用调用。例如，例 7.5 中的两个函数可以改写成：

```
student&    Input_Data(void);
void  Output_Data(const &student);
```

而函数体并不需要改变。这里，将函数 Output_Data（）中的参数说明成 const 的是为了阻止在函数体中修改实参的值，因为在引用调用中函数对形参的访问实际上就是对实参本身的访问。

当函数的参数为指针时，形参与实参的结合方式就属于地址调用。

从以上的例子可以看出，用数组存放数据时必须事先给定数组的大小，当然这个问题可以利用动态内存分配来解决。重要的是对数组中的元素进行删除、插入这样一些常用的表操作是极不方便的。删除数组中的一个元素就得将其后边所有的元素逐一地向前移动一个位置，而插入一个元素时则必须将从插入位置起直到表尾的所有元素逐一地向后移动一个位置。当操作发生在表头即第一个元素上时，移动的数据量将是非常大的。为了解决用数组存放数据处理上的不便，人们构造出了各种各样的数据结构，链表就是其中最基本最重要的数据结构之一。

链表是将一个个数据元素利用指针连结起来的一个表，如图 7-3(a)所示。表中每一个数据元素就叫作链表中的一个结点，结点是由一个数据域 Info 和一个链接域（简称链域）Link 组成的，如图 7-3(b)所示。其中：数据域存放元素的数据；而链域则是一个指针，用来指向下一个结点。头指针 F 是用来存放链表中第一个结点的地址的，最后一个结点链域中的“^”表示该指针是一个空指针，链域为空指针意味着“链表到此结束”。

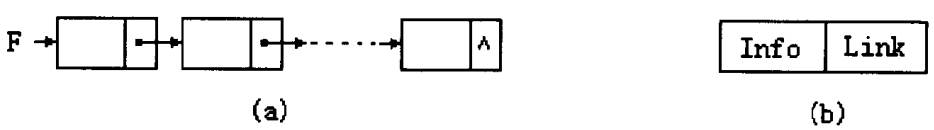

图 7-3　链表及其结点

例 7.6　链表处理。

```
//      EX7_6.H

#if !defined   _EX7_6_H_
#define        _EX7_6_H_

typedef        struct  Person {
       char    *Name;
       char    Sex;
       int     Age;
}      INFO, *PINFO, &RINFO;
```

```
typedef     struct  NODE {          // 结点
        INFO  Info;                 // 数据域
        NODE        *Next;          // 链域
}       *PNODE;

PNODE       CreateList(int);
PNODE       CreateNode(void);
void    Insert(PNODE, PNODE);
PNODE       LookUp(PNODE, RINFO);
PNODE       Delete(PNODE);
void    ShowList(PNODE);
void    DestoryList(PNODE);
void    DestroyNode(PNODE);
INFO  InputInfo(void);
void    OutputInfo(const RINFO);

#endif


//      EX7_6A.CPP

#include      <iostream.h>
#include      <string.h>
#include      "ex7_6.h"

PNODE       CreateList(int n)
// 建立一个长为 n 个结点的链表并返回表头指针
{
        PNODE       np, head = 0, tail;

        while(n --)    {
                np = CreateNode();              // 创建一个新结点
                if(!np)                         // 若创建失败则返回一个空指针
                        return 0;

                if(head == 0)                   // 如果表头为空，则说明新结点应为表中第一个结点
                        head = tail = np;       // 表头指向新结点
                else    {
                        tail->Next = np;        // 将新结点链结到表尾
                        tail = np;              // tail 永远指向表中的尾结点
                }
        }

        return head;                            // 返回表头指针
}

PNODE       CreateNode(void)
// 创建一个新结点。成功时返回指向新结点的指针，失败时返回一个空指针
```

```
{
        PNODE       np;

        np = new NODE;                  // 为新结点分配内存
        if(!np)                         // 若内存分配失败则返回空指针
                return 0;

        np->Info = InputInfo();         // 为新结点输入数据
        np->Next = 0;                   // 新结点链域置为空

        return np;                      // 返回指着新结点的指针
}

void    Insert(PNODE pos, PNODE np)
// 将 np 所指的一个新结点插入到链表中 pos 所指结点的后边
{
        np->Next = pos->Next;           // np 的链域指向 pos 的后继结点
        pos->Next = np;                 // np 所指结点作为 pos 的后继
}

PNODE       LookUp(PNODE list, RINFO ri)
// 在头指针为 list 的链表中查找数据域与 ri 相匹配的结点
{
        int     isEqual(RINFO, RINFO);

        while(list)     {
                if(isEqual(list->Info, ri))
                        return list;    // 找到，返回指向该结点的指针
                list = list->Next;      // 进入下一个结点
        }

        return 0;                       // 未找到，返回空指针
}

PNODE       Delete(PNODE np)
// 从链表中删除指针 np 所指结点的后继结点并返回指向被删结点的指针
{
        PNODE       del;

        if(np == 0 || np->Next == 0)
                return 0;               // 若 np 为一空指针或其所指结点无后继则返回空指针

        del = np->Next;                 // del 指向 np 的后继结点
        np->Next = del->Next;           // 从表中删除该结点
        del->Next = 0;

        return del;                     // 返回指向被删结点的指针
}
```

```
void   ShowList(PNODE   list)
// 显示表头指针为 list 的链表中所有结点数据域的内容
{
      while(list)    {
            OutputInfo(list->Info);
            list = list->Next;              // 进入下一结点
      }
}

void   DestoryList(PNODE list)
// 销毁表头为 list 的链表
{
      PNODE       temp;

      while(list)    {                    // 若结点存在则反复执行
            temp = list;                   // 暂存指向欲删结点的指针
            list = list->Next;             // list 指向下一结点
            DestroyNode(temp);             // 销毁 temp 所指结点
      }
}

void   DestroyNode(PNODE np)
{
      if(!np)
            return;

      delete np->Info.Name;                // 释放 Info.Name 所占的动态内存
      delete np;                           // 释放 np 结点所占的动态内存
}

INFO  InputInfo(void)
// 为一个 INFO 结构变量输入数据
{
      INFO  info;
      char    name[20];

      cout << "Please enter a person's datas:\n";
      cout << "Nmae: ";
      cin >> name;
      info.Name = new char[strlen(name) + 1];        // 为 info.Name 成员分配内存
      strcpy(info.Name, name);                       // 为 info.Name 成员赋值
      cout << "Sex(m/f): ";
      cin >> info.Sex;                               // 为 info.Sex 成员赋值
      cout << "Age: ";
      cin >> info.Age;                               // 为 info.Age 成员赋值

      return info;                                   // 返回 info 的内容
}
```

```
void    OutputInfo(const RINFO ri)
// 输出一个 INFO 结构变量的内容
{
        cout << ri.Name << '\t' << ri.Age << '\t' << ri.Sex << endl;
}

int     isEqual(RINFO ri1, RINFO ri2)
// 判断 ri1 和 ri2 的内容是否相等。本例仅以 Name 成员相同为准
{
        if(strcmp(ri1.Name, ri2.Name) == 0)
                return 1;                       // 相等，返回 1

        return 0;                               // 不相等，返回 0
}


//      EX7_6.CPP

#include      <iostream.h>
#include      <string.h>
#include      "ex7_6.h"

void    main()
{
        PNODE       List = 0, pn, qn;
        INFO  Info;
        char    nm[20];

        List = CreateList(4);                   // 建立一个长为 4 个结点的链表
        ShowList(List);                         // 显示头指针为 List 的链表

        pn = CreateNode();                      // 动态分配一个新结点
        cout << "Do you want this new node insert after who's node? ";
        cin >> nm;                              // 输入插入位置的关键字
        Info.Name = new char[strlen(nm) + 1];
        strcpy(Info.Name, nm);

        if((qn = LookUp(List, Info)) != 0)      // 若找到
                Insert(qn, pn);                 // 则将新结点插入到所找到的结点之后
        ShowList(List);
        cout << "Who's node do you want to delete: ";
        cin >> nm;                              // 输入欲删结点的关键字
        delete Info.Name;
        Info.Name = new char[strlen(nm) + 1];
        strcpy(Info.Name, nm);
        pn = Delete(LookUp(List, Info));        // 删除指定结点
        if(pn)                                  // 若删除成功
                DestroyNode(pn);                // 则将该结点所占之内存返还给系统
        ShowList(List);
```

```
    DestoryList(List);                    // 销毁整个链表
}
```

关于链表的的原理及应用方面的内容已超出了本书的范围，有兴趣的读者可参阅有关“数据结构”方面的资料。

例 7.6 程序的工作原理已在源程序中注释清楚，这里仅作一些补充说明：

1. 由于在结构 Person 中将其成员 Name 说明为字符指针，因此当向该成员赋值前必须为它动态地分配内存，否则将可能造成多个结点的 Name 指向同一个字符串或某个 Name 指向了一个内容已改变了的字符串的现象。正因为如此，在释放一个结点之前必须先释放其成员 Name 所占的内存。

2. 将指向链表中一个结点的指针 p 修改为指向该结点的后继结点时，不得采用对该指针的增量运算——如 p ++——来实现。这是因为每个结点都是通过运算符 new 动态生成的，相继生成的结点不一定在内存中相邻存放。因此，这样的操作必须采用 p = p->Next 的方式来实现。

3. 为了使所设计的链表操作具有一定的通用性，这里将结点的数据域从结点结构中分离了出来，说明成一个独立的结构。若某链表的数据域与本例不同，则只需修改 EX7_6A.CPP 中的与数据域内容直接有关的函数——DestroyNode（），InputInfo（），OutputInfo（）和 isEqual（）——即可。

7.5 位域

在许多应用中，被处理的数据的数值范围往往都很小，比如，常用 1 来表示逻辑真，用 0 表示逻辑假，表示该逻辑值的变量只需 1 位二进制数就可以了。再如，表示一个范围为 0～15 的无符号整数的变量也不需要两个字节，只须 4 位二进制数就足够了。C++语言的位域就是为存储这样的数据而提供的一种节省内存的手段。

所谓位域就是以二进制位为单位长度的一种结构成员。定义位域具有以下的一般形式：

```
struct      struct_name      {
      unsigned    bit_field_name : constant;
};
```

其中：bit_field_name 为结构成员名；constant 是一个整型常量，它的值即为对应成员所需的二进制位数。例如：

```
struct      node {
      char  *name;
      unsigned    sex : 1;
      unsigned    age : 7;
      node  *next;
};
```

这是对例 7.6 中结点 node 的另一种定义。这里，成员 sex 和 age 被说明为位域形式，其

中：sex 占 1 个二进制位，它的值只能是 0 或 1，但这对表示性别已经足够了；而 age 占 7 个二进制位，它的取值范围为 0～127，作为表示年龄的变量，这一取值范围也是可以的了。比较这两种定义方法可以看出：不采用位域时 node 型结构变量需要占用 7 个字节的内存单元；而采用位域后只需要 5 个字节。这是由于成员 sex 和 age 共用一个字节。该结构的存储形式如图 7-4 所示。

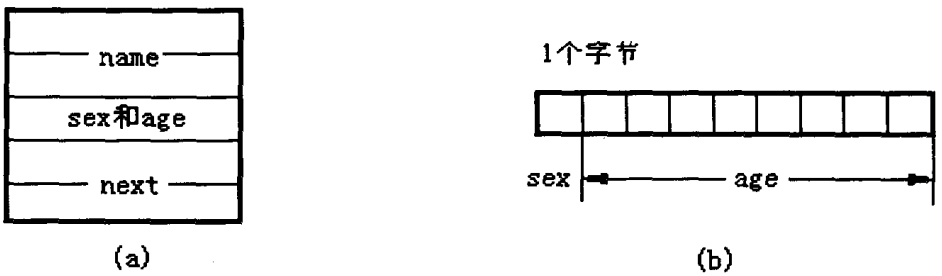

图 7-4　位域的存储形式

包含有位域成员之结构变量的用法与普通结构变量完全一样，即用这里所定义的 node 替换掉头文件 EX7_6.H 中的 node 则原程序基本不用改动，所需做的改动只是将 Out_Dat（）的函数体改为

```
cout << nd.name << '\t' << (nd.sex == 0    ? 'm' : 'f') << '\t' << nd.age << endl;
```

以及将函数 In_Dat（）中的语句

```
cout << "Sex(m/f): ";
```

改为

```
cout << "Sex(0/1): ";
```

即可。这里，用 0 表示男性，1 表示女性。

使用位域时应注意：

1. 位域成员必须说明成 unsigned int 型；
2. 位域成员可以不占满一个字节。比如：

```
struct bit    {
        unsigned    a : 1;
        unsigned    b : 2;
        int     c;
};
```

这里，成员 a 和 b 共占了 3 位，则该字节中其余 5 位闲置不用。

3. 位域成员可以超界存储。比如：

```
struct bit    {
        unsigned    a : 1;
        unsigned    b : 10;
        unsigned    c : 3;
```

```
};
```

则成员 a 和 b 的中 7 位共存于一个字节中，而 b 中另 3 位与 c 共存于一个字节中，整个结构共需两个字节。至于将 b 的那 7 位与 a 共存一个字节，不同的系统有不同的处理。有的系统取 b 的高 7 位而有的系统取其低 7 位。

4. 可以人为地规定某成员从一个新字节开始存放。比如：

```
struct bit    {
        unsigned    a : 1;
        unsigned    b : 10;
        unsinged      : 0;
        unsigned    c : 3;
};
```

其中长度为 0 的成员就是指示将下一个成员 c 从一个新字节开始存放，这样，整个结构就需要 3 个字节。

5. 还可以指定那些位空置不用。比如：

```
struct bit    {
        unsigned    a : 1;
        unsigned      : 3;
        unsigned    c : 3;
};
```

其中无名的成员就指定将成员 a 与 c 之间的 3 位空置不用。

6. 由于不同系统为位域分配空间的策略不尽相同，有的是按成员说明的顺序从字节的高位向低位进行分配，有的则是从低位向高位进行分配。因此，除非程序员对所使用系统的分配策略有充分的了解且不考虑程序的移植性，一般不对位域成员按位操作，对位域成员的访问均应按访问结构成员的方式进行。由于同一原因，上述 4，5 条在实用中也很少使用。

7.6 联合

在实用中有时需要将几个不同类型的数据存放在同一块存储单元中，这就要用到 C++语言的联合类型。

7.6.1 定义联合

关键字 union 用来定义一个联合。定义一个联合具有如下的一般形式：

```
union    union_name    {
    members;
};
```

例如：

```
union un    {
    int    x;
```

```
    char  ch;
    float  real;
};
```

就定义了一个名字为 un 且包含有 3 个成员——整型变量 x，字符型变量 ch 和实型变量 real ——的联合。可见，联合的定义与结构的定义完全一样。

然而，联合与结构是不同的两种数据类型，它们的区别在于：结构中的每个成员分别占有各自的存储单元；而联合中的所有成员共用同一块内存单元。比如，若将 un 定义成一个结构，就需要 7 个字节来存放它的一个变量。联合 un 的大小则为 4 个字节，编译器是按它最大成员 real 的长度为它分配内存的，如图 7-5 所示。

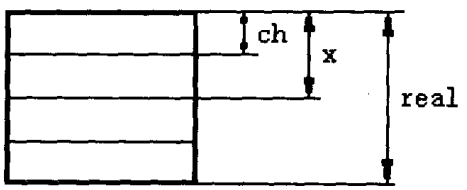

图 7-5　联合的存储形式

由图可见，联合中所有成员具有相同的首地址。

7.6.2　联合的使用

与结构相同，联合的定义只是描述了一个联合类型的组织形式，在程序中须说明属于这种类型的变量。

说明一个联合变量的一般形式为

<storage> union_name var_name<, var_name...>;

例如：

```
un    data;
```

就说明了一个名为 data 的 un 型联合变量。

说明联合变量的同时也可以对其进行初始化，但只允许初始化其中的第一个成员，这是由于联合变量在任一时刻只有一个成员是活动的，且编译器在为联合变量分配内存时将指定的初值以其第一个成员的类型置入所分配的内存中。例如：

```
un    data = {34};
```

就将联合变量 data 中的成员 x 初始化为 34。

除了所有成员共用同一存储空间这一特性外，联合具有与结构完全相同的性质。比如：访问一个联合变量的成员需要使用成员选择运算符“.”或“->”、联合变量可以用作联合或结构的成员（结构变量也可以用作联合的成员）、联合也允许有位域成员、可以说明元素为联合的数组和指向联合类型的指针以及对联合的引用、同类型的联合变量之间可以整体赋值、联合作为函数的参数属于值调用、函数的值可以是一个联合，等等。

例 7.7　联合变量的演示。

```
//      EX7_7.CPP

#include      <iostream.h>
```

```
void   main()
{
       union  un     {
              int     x;
              char    c[2];
       };

       un     data;

       data.x = 0x1234;
       cout << "Address of data.x: " << &data.x << '\t';
       cout << "Address of data.c: " << &data.c << endl;          // A
       cout << "X = " << data.x << '\t' << "C[0] = " << int(data.c[0]) << '\t'
              << "C[1] = " << int(data.c[1]) << endl;
}
```

该程序的输出为：

```
Address of data.x: 0xfff4        Address of data.c: 0xfff4
X = 4660          C[0] = 52   C[1] = 18
```

由程序的输出可见，联合变量 data 的成员 x 和 c 具有同样的首地址。注意程序的行 A 中取地址用的是&data.c 而不是 data.c[0]，这是因为从编器的角度来看，联合 un 只有两个成员——x 和 c，而在程序中对成员 c 进行具体访问时才将它当作一个数组处理。data.x 之值 0x1234 的二进制表示为 00010010 00110100，而基于 Intel 80x86 CPU 的计算机在存储多字节数据时是将低字节存入低地址、高字节存入高地址的，对本例来讲，就是将 00110100 存入地址为 0xfff4 的存储单元中；00010010 存入地址为 0xfff5 的存储单元中，因此，将 c[0]和 c[1]按整数形式输出时就会有以上的结果。

在 C++语言中还可以使用“无名联合”。所谓无名联合是指既无联合名又无联合变量名的联合类型，这种联合类型可以用作结构或联合的成员。关于无名联合的具体例子请参见例 7.9。

7.7 枚举

枚举在数学中是很常用的一种定义一个集合的手段，比如，定义一个 10 以内的偶数的集合可记为：X = {2, 4, 6, 8}。C++语言也提供了类似的定义一个有限整型常数集合的手段，即枚举类型。

7.7.1 枚举类型的定义及其变量的说明

与其他导出数据类型一样，枚举类型也须先行定义。其定义具有以下的一般形式：

enum enum_name{identitiers};

其中：关键字 enum 通知编译器其后边的一个标识符 enum_name 是一个枚举类型的名字；而花括号中的 identifiers 则是一个个用逗号分隔开的标识符，这些标识符称作枚举元素或枚举

常量。比如：

```
enum        weekday{SUN, MON, TUE, WED, THU, FRI, SAT};
```

就定义了一个名为 weekday 的枚举类型，它包含有 7 个元素，这些元素的名子分别为 SUN，MON，…，SAT。

枚举类型的元素是一些标识符常量，编译器在处理一个枚举定义时会自动地为每个元素赋上一个整型常量值，默认的赋值为从 0 开始逐一递增。因此，weekday 的元素 SUN，MON，…，SAT 的值分别为 0，1，2，3，4，5 和 6。这就是将它们称为枚举常量的原因。

可以在定义枚举类型时为枚举常量赋一个初值，以改变系统给定的默认值。比如：

```
enum        bool{TRUE = 1, FALSE = 0};
```

指定的初值必须是一个整型常量。若某个元素没有指定初值，则系统就将它前一个元素的值加 1 后赋给它。比如：

```
enum        weekday{SUN = 7, MON = 1, TUE, WED, THU, FRI, SAT};
```

则各元素的值将为 7，1，2，3，4，5 和 6。

指定的初值也可以是一个负数。比如：

```
enum        sign{NEGATIVE = -1, ZERO, POSITIVE};
```

将使得枚举类型 sign 中 3 个元素的值分别为 - 1，0 和 1。

指定的初值还可以是相同的值。比如：

```
enum        no_mean     {first = 1, second = 1, third = 1};
```

一般情况下，这样的枚举类型没有什么实用价值。

与结构和联合一样，枚举的定义仅仅是定义了一个新的数据类型，必须说明属于这种类型的变量才可以使用这种数据类型。例如：

```
weekday     days;
```

就说明了一个类型为 weekday 的枚举变量 days。

也可以在定义枚举的同时说明枚举变量。比如：

```
enum        sign {NEGATIVE = -1, ZERO, POSITIVE}            sgn;
```

或：

```
enum        NEGATIVE = -1, ZERO, POSITIVE}          sgn;
```

后一个方法定义了一个无名的枚举类型并说明了一个名为 sgn 的枚举变量。使用该方法将使得在程序的其他地方再也不能说明属于这种类型的枚举变量了。

说明枚举变量时也可以对其进行初始化。例如：

```
sign   sgn = ZERO;
```

或：

```
sign   sgn = 0;
```

这两种初始化的作用完全相同。但应注意，后一种初始化方法会引起编译器的警告："将一个整型值赋给了枚举变量"。但是，C++编译器并不认为枚举变量的超界是一个严重的错误，对于如同 sign sgn = 100 这样的初始化它也仅仅是给出上述的警告。所以，在实用中最好使用枚举常量来初始化一个枚举变量。

7.7.2 枚举变量的使用

枚举变量一般只用于赋值运算和关系运算中。

对枚举变量的赋值运算是指将一个枚举常量或一个枚举变量的值赋给枚举变量。比如：

```
sign   sgn, sn;

sgn = ZERO;
sn = sgn;
```

但是，虽然枚举类型的定义限定了枚举变量的取值范围，例如，变量 sgn 的值只应该是 NEGATIVE，ZERO，POSITIVE 其中之一。然而，由于上述的原因，赋值语句：

```
sgn = SUN;
```

也只引起编译器的警告，却不会导致编译失败。因此，在实用中只用变量所属之枚举类型的枚举常量来为变量赋值则是程序员应养成的良好习惯。

枚举变量的关系运算是指比较一个枚举变量与另一个枚举变量或一个枚举常量的关系。比如，设有上述的赋值，则关系表达式 sgn == ZERO 和 sgn == sn 的值均为真。

将枚举变量用于输出语句中时，输出的是一个整数值，而不是它目前所对应的枚举常量的名字。

例 7.8 枚举变量用于输出语句。

```
//      EX7_8.CPP

#include      <iostream.h>

enum  weekday{SUN = 7, MON = 1, TUE, WED, THU, FRI, SAT};

void   main()
{
       weekday      days[] = {MON, RUE, WED, THU, FRI, SAT, SUN};

       for(int i = 0; i < 7; i ++)
              cout << days[i] << '\t';
       cout << endl;
}
```

该程序的输出为：

```
1     2     3     4     5     6     7
```

由程序 EX7_8 的输出可见，一般情况下，输出枚举变量的值没有多大的实用价值。

枚举类型的主要作用是增强编译器对数据类型的检查，改善程序的可读性。

例 7.9 枚举类型的应用。

```
//      EX7_9.CPP

#include      <iostream.h>
#include      <math.h>
#include      <ctype.h>

enum  Format{POINT, SCORE};          // 记分方式之枚举类型（百分制或五级制）
enum  Score{A, B, C, D, E};          // 五级记分之枚举类型

struct  student{                     // 学生记录之结构类型
      int     num;                   // 学号
      union  {                       // 成绩，用无名联合存放
            float   point;           // 百分制成绩
            Score  score;            // 五级记分制成绩
      };
      Format        flag;            // 记分方式标记
};

void   main()
{
      void   In_Dat(student*, int);
      void   Out_Dat(student*, int);

      student        *stu;
      int     n;

      cout << "How many students? ";
      cin >> n;                      // 输入学生人数
      stu = new student[n];          // 建立包含有 n 个学生记录的数组
      In_Dat(stu, n);                // 为数组输入数据
      Out_Dat(n, stu);               // 输出学生数据

      delete []stu;                  // A
}

void   In_Dat(student *s, int n)
// 为一个包含有 n 个元素的 student 型数组 s 输入数据
{
      char   str[4];
      Score  sc[] = {A, B, C, D, E};

      cout << "Please enter student code & score:\n";
      for(int i = 0; i < n; i ++)      {
            cin >> s->num >> str;                  // 输入学号和成绩，成绩以字符串形式保存
            if(isalpha(str[0]))      {             // 成绩为字符形式
                  s->flag = SCORE;                 // 置标记为五级制
```

```
            s->score = sc[tolower(str[0]) - 'a'];   // 存成绩
        }
        else    {                                   // 成绩为百分制
            s->flag = POINT;                        // 置标记为百分制
            s->point = atof(str);                   // 存成绩
        }
        s ++;                                       // 下一个学生
    }
}

void   Out_Dat(int n, student *s)
// 输出一个包含有 n 个元素的 student 型数组 s 中的数据
{
    char   sc[] = {'A', 'B', 'C', 'D', 'E'};

    cout << "\nNo.\tScore\n";
    for(int i = 0; i < n; i ++)      {
        cout << s->num << '\t';                     // 输出学号
        if(s->flag == POINT)                        // 若为百分制记分
            cout << s->point << endl;               // 输出百分制分数
        else
            cout << sc[s->score] <<endl;            // 否则输出五级制分数
        s ++;                                       // 下一个学生
    }
}
```

运行该程序时设用户的响应如下：

How many students: **4**<CR>
Please enter student code & score:
101 78<CR>
102 B<CR>
103 A<CR>
104 66<CR>

则程序的输出为：

No. Score
101 78
102 B
103 A
104 66

该程序的功能与执行过程已在程序中注释清楚，这里仅做一些补充说明：

结构 studnet 中存放成绩的成员是一个包含有两个成员 point 和 score 的无名联合，由于是无名联合，所以它的两个成员就好像是 student 的直接成员，这一点从程序中对这两个成员的访问就可以看的出来。

由于枚举变量的值相当一个整型数，而在五级记分制中分别用字母 A，B，C，D，E 来

表示不同的成绩，因此，在函数 Out_Dat（）中说明了一个元素分别为上述五个字母的字符型数组 sc 供输出成绩时用。请读者自行分析注释为“否则输出五级制分数”语句行是怎样根据枚举变量 score 的值输出相应字母的。

程序中还用到了几个以前没有遇到的 C++语言的库函数，这里对它们作一简单介绍：

库函数 isalpha（）的函数原型在头文件 ctype.h 中说明为

```
int    isalpha(int ch);
```

它的功能是判断变量 ch 中是否是一个 a～z 或 A～Z 的字母，若是则返回一个非零值，否则返回 0。

库函数 tolower（）的函数原型在头文件 ctype.h 中说明为

```
int    tolower(int ch);
```

它的功能是返回变量 ch 中字母的小写形式，不管原字母是何形式。该函数并不改变 ch 本身。

函数 atof（）的函数原型在头文件 math.h 中说明为

```
double      atof(const char *str);
```

它的作用与函数 atoi（）相似，只不过它是将一个数字形式的字符串 str 转换成一个双精度实数。

请读者仔细体会这几个库函数在本程序中所起的作用。

习题

7-1 试画出一个通讯录结构 address_book 型变量的存储形式。

7-2 说明一个表示日期的结构，其成员包括年、月、日。考虑应为该结构定义哪能些操作函数

7-3 说明一个包含有 30 个元素的 address_book 型结构数组。考虑应为该数组定义哪些操作函数

7-4 习题 7-3 只能实现对人数不超过 30 的通讯录的操作。试用一个单链表来存放人数不定的通讯录。（提示：将通讯录结构说明成如下形式来替换例 7.6 中的 Person 结构，并修改对例 7.6 补充说明 3 中所提到的 4 个函数）。

```
typedef     struct {
        unsigned long     Zip;
        char  *Addr;
        unsigned long     Tel;
}       PTA, *PPTA, &RPTA;

typedet     struct {
        char  *Name;
        PTA   Depart;
        PTA   Home;
}       INFO, *PINFO, &RINFO;
```

7-5 将习题 7-2 中的结构成员定义成位域的形式，比较两者所占内存的大小（利用 sizeof 运

算符），试一试在习题 7-2 中所定义的函数是否仍能正常地处理该改动了的结构。

7-6 设计一个记录书籍的结构，其成员包括：书号、书名、作者、出版社和出版时间（利用习题 7-5 中所说明的结构）。利用一个单链表来存放一批书籍，编写函数根据用户输入的关键字输出对应书籍的所有内容。比如，设用户输入的关键字是一个作者的名字，则将该作者所有著作都显示出来。

第 8 章　类和对象

到目前为止，本书所介绍的内容已基本满足编写面向过程的程序所要求的数据类型和程序控制结构。从本章起，将逐步介绍用 C++语言编写面向对象程序所提供的具有抽象性、封装性、继承性和多态性的各种要素。

8.1　概述

面向过程的程序设计方法采用函数（过程）来描述对数据结构类型的操作，但又将函数与其所操作的数据分离开来。作为对现实世界的抽象，函数和它所操作的数据是密切相关、相互依赖的：特定的函数往往是操作特定的数据结构；若数据结构发生改变，则必须改写相应的函数。这种实质上的依赖与形式上的分离使得编写出来的大程序难于控制，严重时甚至导致程序的崩溃。

C++语言中的类体现了 OOP 技术所要求的抽象和封装机制，它将计算机所要处理的对象之属性（数据）和对其进行操作的方法（函数）封装成一个整体。这样，既把自然界中的一类实体（对象）抽象成计算机中的一种具有自主行为的数据类型，同时又把它们的属性隐藏了起来以防止外界的干扰。

数据隐藏可以避免许多维护性问题：如果一个类的数据结构被修改了，除类中操作该数据结构之方法的代码外，软件系统的其余部分是不受影响的。这是因为类中的数据通常在外部是不可见的，外部只能通过公有的接口与它们发生联系。

封装还为继承提供了基础。高层的类通常表示较为一般性的概念，较低层的类则表示更为具体的概念。子类可以从它的父类那里整个地继承其各种行为和属性，同时还可以定义自己所特有的特征。

对象是类的实例，对象之间通过传递消息来实现相互的通信，接收消息的对象有责任响应并处理所收到的消息：按消息名激活本对象内相应的方法并执行其操作、返回适当的结果。发送消息的对象还可以将一条消息发送至多个接收消息的对象中，即，可以把同一消息发送给某一类的父类及其所有子孙类的对象，由系统动态地决定该用哪个类的对象来接收该消息。这就是所谓多态性的概念。

抽象、封装、继承和多态性强有力地支持了复杂大系统的分析与运行，从而克服了传统程序设计方法的不足。

面向对象的程序设计方法不仅大大地提高了程序的可靠性，更重要的是改善了程序代码的重用性。一个为某一目的所设计的类，可以从一个为另一目的而设计的类中派生出来，直接使用父类中的可用部分，而只需添加自身所特有的属性与方法即可。

8.2 类

从技术上讲，类实际上是 C++语言中的一种导出数据类型———一种既包含数据又包含函数的数据类型。

8.2.1 类的说明

与结构、联合等数据类型一样，由于类成员的多变性，所以必须对类先行说明以告知编译器一个具体类的组织形式。说明一个类的一般形式为：

```
class class_name {
        <private:>
                members;
        public:
                members;
        <protected:
                members;>
};
```

其中：关键字 class 指出下面将要说明的是一个类；class_name 是程序员为所说明的类起的名字，它必须是一个有效的 C++标识符；members 为该类中的一个个成员，它们可以是变量——数据成员（属性），也可以是函数——成员函数（方法）；关键字 private 引出的成员叫做私有成员，对私有成员的访问权限制在该类的内部，即只允许该类中的成员函数访问。由于类成员被默认为私有的，所以该关键字可以省略；关键字 public 引出的成员叫做公有成员，公有成员则允许该类以外的函数访问；而关键字 protected 引出的成员则叫做保护成员，保护成员的访问权限将在第 10 章中介绍。

例8.1 说明一个“人”类。

```
//      EX8_1.H

class   Person {
        private:
                char    Name[20];
                unsigned        Age : 7;
                unsigned        Sex : 1;
        public:
                void    Register(const char*, int, char);
                void    GetName(char*);
                int     GetAge();
                char    GetSex();
};
```

任何一个人都必须有姓名、性别和年龄，而类 Person 就体现了对人的抽象，它忽略了一个具体人的身高、体重、肤色、……，等差别，集中了人所具有的共性。同时，在该类中还说明了对一个抽象人之属性进行操作的方法：登录一个人的信息的函数 Register（）；获取

一个人的信息的函数 GetNmae（），GetAge（）和 GetSex（）。该例中，将数据成员说明成私有的以阻止外界对它们的随意访问，而成员函数则说明成公有的以向外界访问类中的数据成员提供统一的“接口”。

本例中的 private 关键字可以省略。

应当说明的是，在一个类中不同访问权限的成员的书写顺序是任意的，比如，上例中的 Person 类也可以改写为：

```
class Person      {
      public:
            void  Register(const char*, int, char);
            void  GetName(char*);
            int   GetAge();
            char  GetSex();
      private:
            char  Name[20];
            unsigned    Age : 7;
            unsigned    Sex : 1;
};
```

在这种情况下，关键字 private 就不能省略，否则编译器将认为该类中的所有成员都是公有的。

类 Person 还可以写成：

```
class Person      {
            char  name[20];
      public:
            void  Register(const char*, int, char);
      private:
            unsigned    Age : 7;
            unsigned    Sex : 1;
      public:
            void  GetName(char*);
            int   GetAge();
            char  GetSex();
};
```

但这样书写形式显得非常混乱。一般情况下应尽量将同一访问权限的成员集中地写在一起。

在文件 EX8_1.H 中仅仅给出了 Person 类中成员函数的函数原型，必须在其他地方给出这些函数的定义。定义成员函数应具有以下的一般形式：

```
type  class_name :: func_name(<arg_list>)
{
     func_body;
}
```

其中：运算符“::”叫做作用域运算符，它指出这时所要定义的函数是一个属于类 class_name

的成员函数。

例8.2 Person 类成员函数的定义。

```
//      EX8_2.CPP

#include      <string.h>
#include      "ex8_1.h"

void    Person :: Register(const char *name, int age, char sex)
{
        strcpy(Name, name);
        Age = age;
        Sex = (sex == 'm' ? 0 : 1);
}

void    Person :: GetName(char *name)
{
        strcpy(name, Name);
}

int     Person :: GetAge()
{
        return Age;
}

char    Person :: GetSex()
{
        return (Sex == 0 ? 'm' : 'f');
}
```

在说明一个类时应注意：

1. 类中任何成员都不得用关键字 extern，auto 和 register 进行修饰。
2. 不得在类说明中对数据成员使用表达式进行初始化。

通常，为了隐藏数据、提供接口，总是将类中的数据成员说明成私有的而将成员函数说明成公有的。然而，一个具体的类中各成员的访问权限应根据实际需要而定。比如，也可以将一个数据成员说明成公有的或将一个成员函数说明成私有的。一般说来，将类中的数据成员说明成公有的将难以保证该成员的安全性，而将类的中成员函数说明成私有的有时还是非常必要的。比如，类中的一个只供其成员函数调用的成员函数就应当说明成私有的。设由于某种原因而不能使用 C++的库函数 strcpy（），于是就有必要由程序员为类 Person 定义一个字符串拷贝函数，这是因为有两个成员函数都需要这样的操作。这时所定义的字符串拷贝函数由于仅供类 Person 的成员函数 Register（）和 GetName（）使用，因此，应该将它说明成私有的。

习惯上将具有全局作用域的类说明放在一个头文件中，将成员函数的定义放在一个另一个源程序文件中，以提高编程的灵活性。对于只具有局部作用域的类的定义，则应放在源程序中。前者说明的类具有外部连接属性而后者说明的类则具有内部连接属性。

8.2.2 类与结构

从类 Person 的说明可以看出：一个类很像一个结构，只不过类的成员可以是函数。事实上，在 C++语言中，结构是类的一个特例。结构也可以包含有函数成员、指定结构成员的访问权限，等等。结构与类的唯一区别就在于：结构成员的访问权限默认为公有的；而类成员的访问权限则默认为私有的。本书后面介绍的关于类的特性若无特别说明均适合于结构。

例8.3 包含有函数成员的结构。

```
//      EX8_3.H

#include      <iostream.h>

struct  Loc    {
        private:
                int    X, Y;
        public:
                void    SetPos(int, int);
                void    GetPos(int&, int&);
};

void    Loc :: SetPos(int x, int y)
{
        X = x;
        Y = y;
}

void    Loc :: GetPos(int &x, int &y)
{
        x = X;
        y = Y;
}
```

本例中，若删去了数据成员 X 和 Y 前边的关键字 private，则编译器将把它们处理成公有的。

在 C++语言中，结构与类具有几乎完全相同的功能，因此在实用中完全可以仅使用结构而不使用类。在习惯上，当仅需要描述数据时，使用结构；而当既需要描述数据又需要描述方法时，则使用类。

8.2.3 内联成员函数

从以上的例子可以看出，类中成员函数的函数体往往都是比较小的，这样的函数最适合写成内联的以提高程序执行的效率。对于成员函数来讲，除了可以像普通函数那样使用关键字 inline 将它定义成内联函数外，还有一种更为简便的方法，这就是在类说明中定义成员函数的函数体。这样说明的成员函数自动成为内联成员函数。

例8.4 将类 Person 的成员函数说明成内联成员函数。

```
//      EX8_4.H
```

```
#include    <string.h>

class  Person {
       private:
              char   Name[20];
              unsigned      Age : 7;
              unsigned      Sex : 1;
       public:
              void   Register(const char*, int, char);
              void   GetName(char *name)
              {
                     strcpy(name, Name);
              }
              int    GetAge()      {return Age;}
              char   GetSex()      {return(Sex == 0 ? 'm' : 'f');}
};

inline void    Person :: Register(const char *name, int age, char sex)
{
       strcpy(Name, name);
       Age = age;
       Sex = (sex == 'm' ? 0 : 1);
}
```

该例中分别用两种方法将类 Person 中的所有成员函数都定义成了内联成员函数，这两种方法的作用是完全相同的。注意：例 8.4 中的程序文件被写成一个头文件（EX8_4.H），这样做是为了使用上的方便。实际上，由于类说明中包含了程序代码，因此将它写成源程序文件（.CPP）也是可以的。

8.3 对象

就如同一个整型变量是类型 int 的一个实例，一个对象就是一个类的实例。从技术上讲，一个对象就是一个具有某种类类型的变量。

8.3.1 对象的说明

与普通变量一样，对象也必须先经说明才可以使用。说明一个对象的一般形式为

<storage> class_name object_name<, object_name...>;

其中：class_name 是一个已存在的类之类名；object_name 是程序员为所说明的对象起的对象名。例如：

```
Person      per1, per2;
```

就说明了两个对象名分别为 per1 和 per2 的 Person 类的对象。

同对结构定义的处理一样，编译器并不对类说明分配内存，它是在程序运行时通过为对

象分配内存来创建对象的。对于上面说明的两个对象，编译器为它们分配的内存理论上如图 8-1 所示。

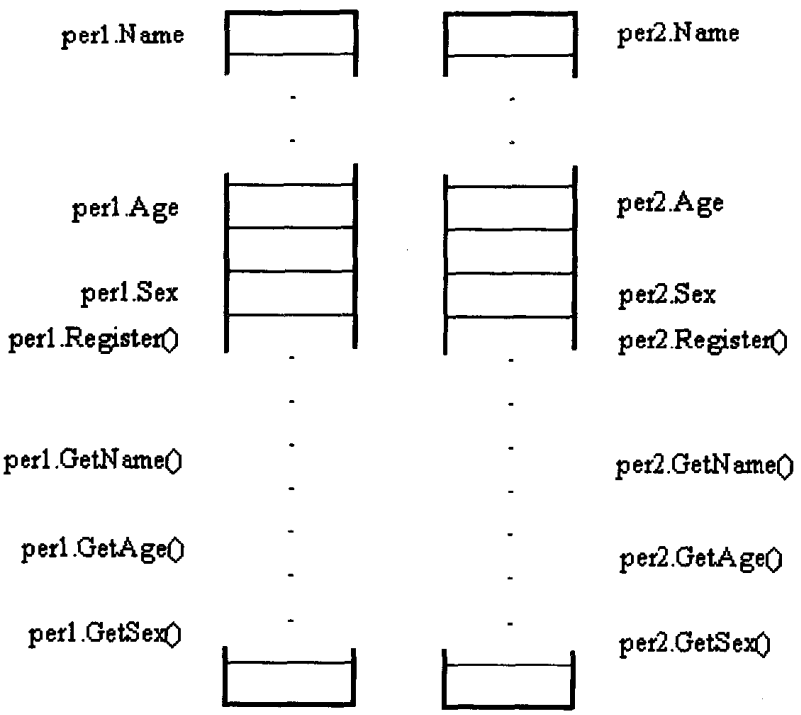

图 8-1　对象 per1 和 per2 的存储形式

由图 8-1 可见，对象在内存中的存储形式与结构变量一样：每一个对象都占据有能保存下其所有成员的存储单元。

不同对象中的数据成员通常具有不同的值，占有不同的存储单元是必须的。然而，一个类中成员函数的代码，无论属于哪个对象，它在整个程序运行期间都不会也不应该发生任何变化。这些相同的代码占有不同的存储单元实在是一种浪费。事实上，C++语言编译器为了节省内存，在创建一个对象时只为数据成员分配内存，而将成员函数一次性地存放在静态存储区，为所有对象所共享。但在概念上仍应将对象按图 8-1 所示理解。

8.3.2　对象的使用

由于类具有与结构几乎完全相同的性质，所以使用一个对象的方法就与使用一个结构变量完全一样：

1. 需使用成员选择运算符“.”访问对象的成员。比如：

```
i = per1.GetAge();
```

就访问了对象 per1 的方法 GetAge（），也就是调用了该成员函数（调用对象的成员函数也叫做向对象发送一个消息），并将其返回值赋给了变量 i。但是，直接访问一个对象中的私有成员则属于非法操作。比如语句：

```
i = per1.Age;
```

将导致一个编译错误。

类通过其公有的成员函数向外部提供了访问其对象中私有成员的公共接口，而这些成员函数往往是经过精心设计和严格调试的，以此保证了其私有数据成员的安全性。

2. 同类对象之间可以整体赋值。例如：

```
per2 = per1;
```

3. 对象用作函数的参数时属于赋值调用；函数可以返回一个对象。

4. 可以在说明类的同时说明对象；可以用初始化列表对对象中的公有数据成员进行初始化。应当指出的是：由于对象中的私有成员在类外是不可见的，所以利用初始化列表对对象中私有成员进行初始化则属于非法操作。例如：

```
class C     {
            int    x;
      //...
}

class D     {
      public:
            int    y
      //...
}

C     c = {3};          // 错误，欲初始化对象 c 中的私有成员 x
D     d = {5};          // 正确，初始化对象 d 中的公有成员 y
```

例 8.5 对象的使用。

```
//      EX8_5.CPP

#include      <iostream.h>
#include      "ex8_4.h"

void    main()
{
      void    Out_Dat(Person);

      char    str[20], c;
      int     ag;
      Person per1, per2;

      cout << "Enter a person's name, age and sex: ";
      cin >> str >> ag >> c;
      per1.Register(str, ag, c);
      per2.Register("Zhang", 34, 'f');

      cout << "Per1:\t";
      Out_Dat(per1);
```

```
        cout << "Per2:\t";
        Out_Dat(per2);
        per2 = per1;
        cout << "\nPer2:\t";
        Out_Dat(per2);
}

void    Out_Dat(Person p)
{
        char    str[20];

        p.GetName(str);
        cout << str << '\t' << p.GetAge() << '\t' << p.GetSex() << endl;
}
```

设用户对该程序的响应为

Enter a person's name, age and sex: **Wang 25 m**<CR>

则程序的输出为：

```
Per1: Wang      25    m
Per2: Zhang     34    f
Per3: Zhou      33    m

Per2: Wang      25    m
```

5. 可以说明指向类类型的指针和对类类型的引用。使用类指针和类引用的主要目的是为了在函数间传递对象时提高程序的运行效率，而不是为了利用地址调用或引用调用来修改对象的属性值。实际上，就是通过指针或引用也不可以访问对象中的私有成员。

6. 可以说明对象数组、指向类类型的指针数组和数组指针。例如：

```
Person      pers[10][5], *pPtr[5], (*p)[3];
```

7. 一个类的对象可以作为另一个类的成员。例如：

```
class X     {
        //...
};

class Y     {
            X     x;
        //...
};
```

这里，类 Y 中的私有成员 x 就是一个类 X 的对象。

8.3.3 类作用域

类说明和结构定义中一对花括号之间的区域叫做类作用域。在类作用域中说明的标识符

其可以见性在类作用域之内。例如：

```
class X    {
      public:
            int    x;
            float  y;
};

int    i = x;        // 错误，x 在此不可见
int    x;            // 正确，这里说明的变量 x 与类 X 中的成员 x 属于不同的作用域
```

类的成员函数无论是内联的还是在类外单独定义的，其函数名都具有类作用域。与块作用域一样，类作用域中的标识符将屏蔽包含该类作用域之作用域中的同名标识符。比如：

```
class X    {
            int    x;
      public:
            int    fun();        // 类 X 的成员函数说明
};

int    x;
int    fun();                   // 全局函数 fun（）的说明
int    X :: fun()               // 类 X 之 fun（）方法的定义
{
      :: x ++;                  // 全局变量 x
      return x;                 // 类成员 x
}
```

类说明也分为引用性说明和定义性说明。在说明一个类时没有明确地给出该类的组织形式而仅仅给出了类名就属于引用性说明，比如，函数说明中的类类型参数和函数值就是对类的引用性说明。还有一种叫做超前说明（Forward Declare）的引用性说明，这种说明通常用在有交叉引用的类说明中。超前说明的类不能用来建立对象，它只能用来说明函数的形参、函数的值、指针和引用。例如：

```
class X;                    // 超前说明

class Y    {
            X    *px;
      //...
};

class X    {                // 定义性说明
            Y    y;
      //...
};
```

这里，类 X 中包含有类 Y 对象的成员，所以类 Y 必须在说明类 X 之前就有定义；但类 Y 中

又包含有一个指向类 X 的指针成员（这种现象就属于交叉引用），因此必须对类 X 进行超前说明。这一点同样适合于结构。

8.4 成员函数的重载

类中的成员函数与普通函数一样，也可以重载，成员函数也可以带有缺省参数。

例 8.6 一个线性表类。

```
//      EX8_6.H

class   Linear_List   {
            int     *List;                                  // 指向表头的指针
            unsigned        nMax;                           // 表的最大长度
            unsigned        nElem;                          // 表中实际元素个数
        public:
            void    Init(int n = 10)                        // 初始化线性表
            {
                List = new int[n];
                nMax = n;
                nElem = 0;
            }
            int     Elem(int);
            int     &Elem(unsigned n)   {return List[n];}   // 返回表中下标为 n 的元素值
            unsigned        Elem() {return nElem;}          // 返回表中实际元素个数
            unsigned        Max()  {return nMax;}           // 返回线性表的最大长度
            void    Print();
            void    Destroy()       {delete List;}          // 销毁线性表
};


//      EX8_6A.CPP

#include      <iostream.h>
#include      "ex8_6.h"

int     Linear_List :: Elem(int elem)
// 将 elem 添加到表尾，返回非零值表示操作成功
{
        if(nElem < nMax)    {
            List[nElem ++] = elem;

            return nElem;
        }

        return 0;               // 表已满，操作失败
}

void    Linear_List :: Print()
```

```
// 输出表中所有元素的值
{
        for(int i = 0; i < nElem; i ++)
                cout << List[i] << endl;
}


//      EX8_6.CPP

#include      <iostream.h>
#include      "ex8_6.h"

void    main()
{
        Linear_List    list, list1;

        list.Init(20);
        list1.Init();

        for(int i = 1; i <= 5; i ++)
                list.Elem(i);

        cout << "The length of list1 is " << list1.Max()
                <<" and there are " << list1.Elem() << " elements in it.\n";
        cout << "The length of list is " << list.Max()
             <<" and there are " << list.Elem() << " elements in it.\n";
        list.Print();

        list.Elem(3u) = 100;                                          // A
        list.Elem(20);
        cout << "\nNow, there are " << list.Elem() << " elements in list:\n";
        list.Print();
        cout << "The last element is " << list.Elem(5u) << endl;      // B
        list.Destroy();                                               // C
        list1.Destroy();
}
```

该程序的输出为：

```
The length of list1 is 10 and there are 0 elements in it.
The length of list is 20 and there are 5 elements in it.
1
2
3
4
5

Now, there are 6 elements in list:
1
2
3
```

```
100
5
20
The last element is 20
```

类 Linear_List 中有 3 个重载了的公有成员函数：方法 int Elem(int)的作用是将一个新元素插入到线性表的尾部；方法 int &Elem(unsigned) 的作用是返回对指定下标元素的引用，将该函数设计成返回引用的好处是既可以对该元素进行赋值（如程序中行 A 所示）又可以取得该元素的值（如程序中行 B 所示）；方法 int Elem（）的作用是返回线性表中元素的个数。从该例可以看出，在程序中通过方法名向类的一个对象发送一个消息，比如 list.Elem（），而如何响应一个消息则由对象根据消息的性质自行处理。

注意成员函数 Destroy（）的作用：当类 Linear_List 的一个对象的生存期结束时，系统会自动地收回该对象所占有的内存单元。然而，一个对象所代表的线性表本身则是通过运算符 new 动态地分配的，对象中的 List 成员仅是一个指向这块内存首地址指针。对象消失后，成员 List 所占的存储单元被释放了，而它所指的动态分配内存单元系统并不负责释放，该函数就是负责释放这块存储单元的。程序中的行 C 就是在两个对象消亡前释放它们所占的动态内存的。

函数重载机制从一个方面体现了对象间通过消息传递信息的多态性。

8.5 this 指针

当一个对象接收到一个消息时，系统会自动地向它传递一个隐含的参数——指向该对象的指针。该指针称为“this”指针，这是由于在程序中可以通过关键字 this 访问该指针。

可以这样来理解 this 指针：前边所有例子中成员函数的定义实际上是一种缩写形式，以类 Linear_List 的成员函数 int Elem(int)的定义为例，它完整形式应为：

```
int    Linear_List :: Elem(int elem)
{
       if(this->nElem < this->nMax) {
              this->List[this->nElem++] = elem;

              return this->nElem;
       }

       return 0;
}
```

这里的 this 指针具有如下的缺省说明：

```
Linear_List *const       this;
```

在例 8.6 中，当程序执行语句

```
list.Elem(i);
```

时，系统对 this 指针进行了如下的缺省赋值：

```
this = &list;
```

这样，成员函数就可以对对象 list 的数据成员而不是对其他 Linear_List 的对象的数据成员进行操作。

仅从以上各例看来，this 指针完全由系统自动地掌握，程序员似乎不需要过多地注意它。实际上，在程序设计中，this 指针常常具有不可替代的作用。例如，为了能将一个线性表赋给另一个线性表（注意：尽管两个同类的对象可以直接进行赋值操作，但对于像 Linear_List 这样定义的类，若将其一个对象直接赋给另一个对象，将会导致一个严重的逻辑错误。关于这一点，将在第 9 章的 9.3 中做进一步的介绍），可以为类 Linear_List 增加一个赋值成员函数：

```
void  Assign(Linear_List&);
```

该函数的定义为：

```
void  Linear_List :: Assign(Linear_List &l)
{
      if(&l == this)                          // A
            return;

      delete l.List;                          // 删除目标对象所占的动态内存

      nMax = l.nMax;
      nElem = l.nElem;
      List = new int[nMax];                   // 重新为目标对象分配内存
      for(int i = 0; i < nMax; i++)
            List[i] = l.List[i];
}
```

这里，行 A 的 if 语句是为了防止形如：

```
list.Assign(list);
```

这样的自身赋值。在这种情况下，就只能使用 this 指针。

习题

8-1　现实世界中的物理实体和逻辑概念均可以看作是对象。分析你周围的一个对象，试用 C++语言的类来描述它。

8-2　将第 7 章中的通讯录结构改写为类，考虑应为它定义哪些成员函数。

8-3　将例 8.5 中的类 Person 改写为结构，使得例 8.5 仍能正常运行。

8-4　将习题 7-2 中的日期结构改写为类。

8-5　将习题 7-6 中的书籍结构改写为类。

第 9 章　构造函数和析构函数

有几类特殊的成员函数，它们决定了类的对象如何创建、初始化、复制和撤消。它们就是本章将要介绍的构造函数和析构函数。

9.1　构造函数

构造函数（Constructor）定义了创建对象的方法，提供了初始化对象的一种简便手段。

9.1.1　定义构造函数

在类中定义构造函数具有以下一般形式：

```
class_name(<arg_list>)
{
    func_body;
}
```

当构造函数在类外定义时，与其他成员函数一样，须加类名与作用域运算符：

```
class_name :: class_name(<arg_list>)
{
    func_body;
}
```

本书在以后的论述中仅用这第二种形式给出定义成员函数的一般形式，不再区分是在类中还是在类外进行定义。

构造函数也允许重载。

例 9.1　为类 Person 增加构造函数。

```
//      EX9_1.H

#include      <string.h>

class   Person {
        private:
                char    Name[20];
                unsigned        Age : 7;
                unsigned        Sex : 1;
        public:
                Person(char *name, int age, char sex)                    // 构造函数
```

```
    {
        strcpy(Name, name);
        Age = age;
        Sex = (sex == 'm' ? 0 : 1);
    }
    Person()    {};                                    // 重载的构造函数
    void  SetNmae(char *name)      {strcpy(Name, name);}
    void  SetAge(int age)      {Age = age;}
    void  SegSex(char sex)     {Sex = (sex == 'm' ? 0 : 1);}
    void  GetName(char *name)      {strcpy(name, Name);}
    int   GetAge()     {return Age;}
    char  GetSex()     {return(Sex == 0 ? 'm' : 'f');}
};
```

在本例中，为类 Person 增加了两个重载了的成员函数——构造函数。其中一个函数体为空，它相当一个函数体为一条空语句的函数。

定义构造函数时应注意：

1. 构造函数的函数名与类名完全相同。

2. 构造函数可以带有零个或多个参数；但不得有返回值，甚至连关键字 void 也不允许有。

3. 一个类可以定义零个或多个构造函数，若定义了多个构造函数，则必须符合 C++语言对重载函数的规定。

9.1.2 构造函数与对象初始化

当创建立一个对象时，系统将自动地调用对象所属类的构造函数。若定义了带有参数的构造函数，就可以在说明对象时利用实参对对象运行初始化。例如：

```
Person     per1("Zhang", 23, f), per2;
```

当遇到这样的说明时，编译器就调用构造函数

```
Person :: Person(char*, int, char);
```

来创建对象 per1 并用实参初始化它的数据成员；并调用构造函数

```
Person :: Person();
```

来创建对象 per2。但由于对象 per2 没有经初始化，它的数据成员中都是一些不确定的值，因此在类说明中又增加了 3 个设置属性值的成员函数：设置姓名的方法 SetName（），设置年龄的方法 SetAge（）和设置性别的方法 SetSex（）。它们不仅可以为未初始化的对象赋值，而且还可以用来修改对应属性的值。

下面通过一个例子来演示构造函数的执行过程。

例 9.2 构造函数的执行过程。

```
//      EX9_2.H

#include      <iostream.h>
```

```
class   Demo {
            int     x, y;
        public:
            Demo(int a, int b)
            {
                x = a;
                y = b;
                cout << "Constructor Demo(int, int) be called...\n";
            }
            Demo()
            {
                cout << "Constructor Demo() be called...\n";
            }
            void    Show()
            {
                cout << "X = " << x << '\t' << "Y = " << y << endl;
            }
};


//      EX9_2.CPP

#include      "ex9_2.h"

void    main()
{
        Demo  d1(3, 5);                 // A

        d1.Show();

        Demo d2;                        // B

        d2.Show();
}
```

该程序的输出为：

```
Constructor Demo(int, int) be called...
X = 3 Y = 5
Constructor Demo() be called...
X = 946     Y = 928
```

由例 9.2 的输出可以看出：当执行语句 A 时，程序调用了构造函数 Demo（int, int)；而在执行语句 B 时，则调用了构造函数 Demo（)。由于对象 d2 未经初始化，所以输出中最后一行的值是一些随机值。

应当说明的是：遇到带有关键字 extern 修饰的外部对象说明时不调用构造函数，因为该说明属于对外部已经定义过的对象的引用性说明。

9.1.3 构造函数和 new 运算符

运算符 new 可以用来建立生存期可控的对象。由于类从本质上讲就是一个数据类型，因

此用 new 创建一个对象的方法与建立动态变量的用法相似。然而，用 new 创建一个对象时将调用类中相应的构造函数。

例 9.3 用运算符 new 创建动态对象。

```
//      EX9_3.CPP

#include      "ex9_2.h"

void   main()
{
       Demo  *dPtr1 = new Demo(3, 5);

       dPtr1->Show();

       Demo  *dPtr2 = new Demo;

       dPtr2->Show();

       delete dPtr1;
       delete dPtr2;
}
```

该程序的输出与例 9.2 完全相同。

注意：用 new 动态建立的对象必须在不再需要它时利用 delete 运算符将其释放，这是程序员的责任。

动态建立一个对象常常是为了临时的需要或是为了控制一个对象的生存期。

例 9.4 例 9.3 的另一版本。

```
//      EX9_4.CPP

#include "ex9_2.h"

void   main()
{
       Demo *dPtr = new Demo(3, 5);

       dPtr->Show();
       delete dPtr;

       dPtr = new Demo;

       dPtr->Show();
       delete dPtr;
}
```

在该程序运行时，任一时刻仅有一个 Demo 类的对象存在，并且总是一个由指针 dPtr 指着的无名对象。利用这种技术，可以有效地利用内存。

9.1.4 缺省的构造函数

以上多次提到：系统在创建一个对象时将自动地调用对象所属类的构造函数。那么，为什么在第 8 章中所说明的类中都没有定义构造函数而它们的对象却都能正常建立呢？这是因

为在一个类中若没有定义任何构造函数的话，系统会为自动地为它定义一个形如：

```
class_name :: class_name()    {}
```

的构造函数。这种不带任何参数且函数体为空的构造函数就叫作缺省的构造函数。但若在类说明中定义了任何构造函数的话，系统将不再提供缺省的构造函数。在这种情况下，若程序员没有显式地定义缺省的构造函数，则该类就不存在缺省的构造函数。这时，对于形如：

```
class_nams object;
```

的对象说明，将由于编译器找不到缺省的构造函数而导致编译失败，除非在类中定义了一个所有参数都带有缺省值的构造函数，编译器将使用该构造函数来匹配这一说明。例如，设有类说明：

```
class X    {
           int    x, y;
      public:
           X(int a, int b);
};
```

则对象说明：

```
X    x;
```

将导致一个编译错误。若将上述构造函数改为

```
X(int a = 0, int b = 0);
```

则编译器将调用它来创建对象 x，这时，该对象的数据成员 x 和 y 的值均被初始化为 0。

由于上述原因，建议程序员在说明一个类时，若定义了构造函数，则应该显式地定义一个缺省的构造函数。特别是在需要说明对象数组时，更应如此。

例 9.5 对象数组。

```
//     EX9_5.CPP

#include     "ex9_2.h"

void   main()
{
      Demo cArr[5];                                          // A
      Demo dArr[3] = {Demo(1, 2), Demo(3, 4), Demo(5, 6)};   // B

      for(int i = 0; i < 3; i ++)
            dArr[i].Show();
}
```

该程序的输出为：

```
Constructor Demo() be called...
Constructor Demo() be called...
Constructor Demo() be called...
```

```
Constructor Demo() be called...
Constructor Demo() be called...
Constructor Demo(int, int) be called...
Constructor Demo(int, int) be called...
Constructor Demo(int, int) be called...
X = 1 Y = 2
X = 3 Y = 4
X = 5 Y = 6
```

若在头文件 EX9_2.H 中将缺省的构造函数的定义删去，则在编译行 A 时由于找不到缺省的构造函数而导致编译失败。而行 B 由于调用了构造函数 Demo（int, int）来初始化数组的元素，所以编译器在创建数组 dArr 时，不再需要调用缺省的构造函数。由此可见，除非在程序中定义对象数组时总要调用有参数的构造函数初始化数组元素，或说明类时一个构造函数也没有定义，否则类中缺省的构造函数必须显式地定义。

应当指出的是：在定义构造函数时，一定要注意不要引起对象说明时的二义性。例如，在类 Demo 的说明中将构造函数定义成带有缺省参数的函数：

```
Demo :: Demo(int a = 0; b = 0);
```

则编译器在遇到对象说明：

```
Demo      d;
```

时，就不知道是应该调用构造函数 Demo（）还是调用构造函数 Demo（int, int）来创建对象 d，这就产生了二义性。

若在类中存在一个所有参数都指定了缺省值的构造函数，则该构造函数就可以代替缺省的构造函数，因此就没有必要也不应该再定义缺省的构造函数了。

9.2 析构函数

与构造函数相对应，析构函数（Destructor）是用来撤消一个对象的。

9.2.1 定义析构函数

定义一个析构函数具有以下的一般形式：

```
class_name :: ~class_name()
{
    func_boey;
}
```

当一个对象的生存期结束时，系统将自动地调用析构函数来撤消该对象，返还它所占据的内存块。

例 9.6 析构函数的执行过程。

```
//    EX9_6.H
```

```
#include      <iostream.h>

class  Demo  {
            int     x, y;
      public:
            Demo(int a, int b)
            {
                  x = a;
                  y = b;
            }
            Demo()       {}
            ~Demo()                        // 析构函数
            {
                  cout << "Destructor be called...\n";
            }
            void   Show()
            {
                  cout << "X = " << x << '\t' << "Y = " << y << endl;
            }
};


//      EX9_6.CPP

#include      "ex9_6.h"

Demo  d(10, 20);

void   main()
{
      void fun();

      Demo  d1(3, 5);

      d1.Show();
      fun();
      cout << "In main()...\n";
      d.Show();
}

void   fun()
{
      cout << "In fun()...\n";

      Demo  d2[2] = {Demo(1, 2), Demo(3, 4)};

      for(int i = 0; i < 3; i ++)
            d2[i].Show();
}
```

该程序的输出为：

```
X = 3 Y = 5
In fun()...
X = 1 Y = 2
X = 3 Y = 4
Destructor be called...
Destructor be called...
In main()...
X = 10      Y = 20
Destructor be called...
Destructor be called...
```

从程序的输出可见，当函数 fun（）的执行结束时，在函数内部说明的对象数组 d2 的生存期也随之结束，系统就调用析构函数将其撤消。当整个程序结束时，系统首先调用析构函数撤消主函数中的局部对象 d1，然后调用析构函数撤消全局对象 d。

定义析构函数时应注意：

1. 析构函数名与类名相同，只是它的前边须冠以波浪号“~ ”以与构造函数区别开来。

2. 析构函数不得带有任何参数，即其参数表必须为空，即使关键字 void 也不允许有。因而，析构函数也就不能重载。

3. 析构函数不得返回任何值，即使关键字 void 也不允许有。

4. 在析构函数中不得调用 C++的库函数 exit（），这是因为该函数会做一些清理工作，包括撤消对象，因此，在析构函数中调用该函数时，该函数为撤消对象又会调用析构函数，从而形成无限递归。如果必须在析构函数中终止整个程序的运行，则应调用库函数 abort（）。

9.2.2 析构函数和 delete 运算符

先看一个例子。

例 9.7 析构函数与动态对象数组。

```
//      EX9_7.CPP

#include      "ex9_6.h"

void   main()
{
       Demo  *dPtr = new Demo[3];
}
```

运行该程序会发现它没有任何输出，说明在该程序中没有调用类 Demo 的析构函数。这是因为在该程序中没有说明类 Demo 的对象，所创建的对象数组是利用运算符 new 动态生成的，而撤消动态数组（无论是否对象数组）则是程序员的责任。下例是对例 9.7 的改正：

例 9.8 析构函数与动态对象数组的正确操作。

```
//      EX9_8.CPP
```

```
#include      "ex9_6.h"

void   main()
{
       Demo  *dPtr = new Demo[3];

       delete [3]dPtr;                    // A
}
```

该程序的输出为：

```
Destructor be called...
Destructor be called...
Destructor be called...
```

注意程序中行 A 返还动态分配的内存时的书写形式：从原则上讲，利用 delete 运算符来撤消一个动态数组时，应当在被撤消的指针前用一个方括号，指出欲撤消的是一个数组，并且在方括号中用一个整型数据通知系统撤消的是包含了多少个元素的动态数组。然而在具体实现中，对于由基本数据类型（包括无成员函数的结构）构成的动态数组，撤消时可以省去方括号及其中的数据。但对于动态对象数组，则不能有此省略，即必须有这一对方括号，以便系统能够正确地为每一个对象调用它们的析构函数。

比如，若将例 9.8 中的行 A 改写为

```
delete dPtr;
```

程序的输出则变为：

```
Destructor be called...
Null pointer assignment
```

这是因为执行该语句时，由于指针 dPtr 是一个 Demo 型的指针，则在释放该指针所指内存时须先调用析构函数撤消它所指的对象，这时实际上仅对 dPtr[0]实施了析构操作。当程序结束时，系统通常还要做一些清理工作，它发现还有一些动态分配的内存没有返还给操作系统，而当它企图返还这部分内存时，又发现指向这块内存首地址的指针已被删除，因而发出一个“对空指针赋值”的错误信息。因此，利用 new 动态分配的对象数组必须正确地利用 delete 将其所占的内存返还给系统。

需要指出的是：无论是基本数据类型动态数组还是动态对象数组，方括号中的数据并非必需的。事实上，对于上例的行 A，有的编译器还会给出一个警告信息：

```
Size of array is ignored
```

即编译器将忽略该数据。因此，在实用中，习惯上在 delete 一个由基本数据类型构成的动态数组时，不加方括号；而在 delete 一个动态对象数组时，加一对空的方括号。

9.2.3 缺省的析构函数

与构造函数一样，若在类说明中没有显式地定义析构函数，则系统将会自动地产生一个形如：

```
class_name :: ~class_name()  {}
```

的析构函数，该函数体为空的析构函数就叫做缺省的析构函数。

分析头文件 EX9_6.H 中的析构函数可以看出，它实际上就是一个缺省的析构函数，其函数体中的语句仅仅是为演示用的，在实用中该语句必须删去。那么，是否在程序中就没有必要再定义析构函数了呢？实际上，在实用中还经常需要重新定义析构函数，以在撤消对象时做一些必要的处理。

例 9.9 线性表类的另一版本。

```
//      EX9_9.H

#include      <iostream.h>

class  Linear_List   {
            int     *List;
            unsigned      nMax;
            unsigned      nElem;
       public:
            Linear_List(int n = 10)
            {
                  List = new int[n];
                  nMax = n;
                  nElem = 0;
            }
            int     AppendElem(int);
            int     &Elem(unsigned n)   {return List[n];}
            unsigned      ElemNum()   {return nElem;}
            unsigned      ListLen()     {return nMax;}
            void   Print();
            ~Linear_List()        {delete List;}
};

int     Linear_List :: AppendElem(int elem)
{
       if(nElem < nMax)    {
            List[nElem ++] = elem;
            return nElem;
       }
       return 0;
}

void   Linear_List :: Print()
{
       for(int i = 0; i < nElem; i ++)
            cout << List[i] << endl;
}
```

可以看出，例 9.9 是对例 8.6 中类 Linear_List 说明的修正：将成员函数 Init（）用带有缺省参数的构造函数代替；将成员函数 Destroy（）用析构函数代替。另外，还改变了其它

几个成员函数的函数名以改善程序的可读性。经过这样的改造，对类 Linear_List 的使用将变得简洁方便，清晰易读。

例 9.10 使用改造后的类 Linear_List。

```
//      EX9_10.CPP

#include      <iostream.h>
#include      "ex9_9.h"

void   main()
{
       Linear_List    list(20), list1;

       for(int i = 1; i <= 5; i ++)
              list.AppendElem(i);

       cout << "The length of list1 is " << list1.ListLen()
              <<" and there are " << list1.ElemNum() << " elements in it.\n";
       cout << "The length of list is " << list.ListLen()
            <<" and there are " << list.ElemNum() << " elements in it.\n";
       list.Print();
       list.Elem(3u) = 100;
       list.AppendElem(20);
       cout << "\nNow, there are " << list.ElemNum() << " elements in list:\n";
       list.Print();
       cout << "The last element is " << list.Elem(5u) << endl;
}
```

该程序的输出与例 8.6 的输出完全相同。比较源程序 EX8_6.CPP 和 EX9_10.CPP，显然后者要比前者来的简洁。实际上，后者执行起来也要比前者快捷，其原因请读者自行分析。

9.3 拷贝初始化构造函数

所谓拷贝初始化构造函数是指形如：

class_name :: class_name(class_name&);

的构造函数，即其参数为对该类类型之引用的构造函数。

在利用拷贝初始化构造函数来创建一个对象时，须用一个已存在的同类对象或引用作为实参。

例 9.11 含有拷贝初始化构造函数的类。

```
//      EX9_11.H

#include      <iostream.h>

class  Demo {
              int      x, y;
       public:
```

```
        Demo(int a = 0, int b = 0)
        {
                x = a;
                y = b;
        }
        Demo(Demo &d)
        {
                x = d.x;
                y = d.y;
        }
        Demo()        {}
        void    Show()
        {
                cout << "X = " << x << '\t' << "Y = " << y << endl;
        }
};


//      EX9_11.CPP

#include      "ex9_11.h"

void    main()
{
        Demo  d1(3, 5), d2(d1);

        d1.Show();
        d2.Show();
}
```

该程序的输出为：

```
X = 3 Y = 5
X = 3 Y = 5
```

EX9_11 程序执行时，对象 d2 通过将对象 d1 做实参调用拷贝初始化构造函数来创建，根据拷贝初始化构造函数的定义，d2 的数据成员取得了与 d1 数据成员相同的值。

然而，若删去 EX9_11.H 中的拷贝初始化构造函数，则例 9.11 程序不仅能顺利地编译通过而且输出结果也是正确的。这是由于若类中没有显式地定义拷贝初始化构造函数的话，当编译器遇见形如：

```
class_name obj1(obj2);
```

的对象说明时，会自动地产生一个缺省的拷贝初始化构造函数，并利用对象 obj2 所有数据成员的值用来初始化对象 obj1 对应的数据成员。这里 obj2 必须是一个已经说明过的类 class_name 的对象。

那么，还有必要显式地定义类的拷贝初始化构造函数吗？有必要。比如，在说明一个对象时，只希望用一个已存在的同类对象的部分属性来进行初始化：

```
Demo :: Demo(Demo &d)      {x = d.x;}
```

则该拷贝初始化构造函数就必须显式地定义。但是，从拷贝初始化构造函数的说明形式可以看出：拷贝初始化构造函数是不可以重载的，因此，这种用法实际上极少使用。问题在于，对于类似类 Linear_List 的对象，若利用缺省的拷贝初始化构造函数来创建一个对象将会出现严重问题，例如：

```
Linear_List list1(list);
```

设对象 list 已经存在，则系统将利用缺省的拷贝初始化构造函数来创建对象 list1，并做如下的赋值：

```
list1.List = list.List;
list1.nMax = list.nMax;
list1.nElem = list.nElem;
```

这样，对象 list 和 list1 的成员 List 就指向了同一块内存的首址。当这两个对象中任一个的生存期结束时，它的析构函数就将这块内存释放，这就导致另一个对象的成员 List 指向一块已释放了的内存的首地址。当第二个对象的生存期结束时，它的析构函数又要释放这一已释放了的内存，这将导致一个运行时间错误（Run time error）。为解决这一问题，就有必要为类 Linear_List 显式地定义拷贝初始化构造函数，指定它的操作方式：

```
Linear_List :: Linear_List(Linear_List &l)
{
    nMax = l.nMax;
    nElem = l.nElem;
    List = new int[nMax];
    for(int i = 0; i < nMax; i ++)
        List[i] = l.List[i];
}
```

该拷贝初始化构造函数很像第 8 章中的 Assign（）函数，但该函数中少了两条 if 语句。这是因为：第一，一个编译成功的程序中不可能出现形如：

```
Linear_List list(list);
```

这样用一个正在创建的对象来初始化一个正在创建的对象的说明语句，因此也就没有必要判断参数与 this 指针的关系；第二，由于构造函数是用来创建一个对象的，尚未创建的对象当然不存在占据动态内存的问题，因此也不存在需要删除动态内存的可能性。

例 9.12 类 Person 的改进。

```
//      EX9_12.H

#if !defined   _EX9_12_H_
#define        _EX9_12_H_

#include       <iostream.h>

class  Person {
       private:
```

```
        char    *Name;
        unsigned        Age : 7;
        unsigned        Sex : 1;
        void    CopyName(char*);
    public:
        Person(char*, int, char);
        Person(Person&);
        Person()
        {
            Name = 0;
            Age = 0;
            Sex = 0;
        }
        ~Person()       {delete Name;}
        void    SetName(char*);
        void    SetAge(int age)         {Age = age;}
        void    SetSex(char sex)        {Sex = (sex == 'm' ? 0 : 1);}
        char    *GetName(char*);
        int     GetAge()        {return Age;}
        char    GetSex()        {return(Sex == 0 ? 'm' : 'f');}
        void    Show();
};

#endif


//      EX9_12.CPP

#include      <string.h>
#include      "ex9_12.h"

void    Person :: CopyName(char *str)
{
    Name = new char[strlen(str) + 1];
    strcpy(Name, str);
}

Person :: Person(char *name, int age, char sex)
{
    CopyName(name);
    Age = age;
    Sex = (sex == 'm' ? 0 : 1);
}

Person :: Person(Person &p)
{
    CopyName(p.Name);
    Age = p.Age;
    Sex = p.Sex;
}
```

```
void Person :: SetName(char *name)
{
        if(Name)
                delete Name;
        CopyName(name);
}

char    *Person :: GetName(char *str)
{
        strcpy(str, Name);

        return str;
}

void    Person :: Show()
{
        char    str[81];

        cout << "Name: " << GetName(str) << '\t' << "Age: " << GetAge()
                << '\t' << "Sex: " << GetSex() << endl;
}
```

改造后的类 Person 由于其 Name 成员被定义成一个字符型指针，这样就可以最有效地使用内存。另外，由于多个成员函数中都有一些相同的操作，因而在类中定义了一个私有成员函数 CopyName（）以供它们调用。

需要指出的是：这里定义了类 Person 缺省的构造函数的函数体。在该函数体中，将成员 Name 置为空串，Age 和 Sex 的值置为 0，这将使得所有由该构造函数所创建的对象具有统一的值。然而，这样定义缺省构造函数的意义还不在于此，关键在于将字符指针型成员 Name 置为空，以适应其他的成员函数。例如，在成员函数 SetName（）中，存在有对 Name 的测试：若 Name 不空，则释放它所指向的内存块。如果一个对象是通过调用无函数体的缺省构造函数所创建的，则其数据成员的值均是一些随机值，这将使得上述的测试结果为真，从而使得程序执行语句：

```
delete Name;
```

但在这以前并没有用运算符 new 为 Name 分配内存，上述语句释放的将是一块未知的内存块。若 Name 的值正好是系统正在使用的一块关键内存中的一个地址，则对 Name 的 delete 操作将可能导致系统崩溃。因此，为缺省的构造函数定义有意义的函数体是对所设计的类的安全性和可靠性的重要保证。

这里还需要强调说明成员函数 GetName（）的设计。事实上该函数完全可以定义成如下的形式：

```
char   *Person :: GetName()
{
        return Name;
}
```

然而这样的定义却存在一个严重的漏洞，那就是程序可以通过该函数所返回的地址来修改对象中私有成员 Name 的内容，从而破坏了类的封装性。因此，用一个字符型数组作为参数，将对象的 Name 成员的内容复制出来就可以有效地保障对象中数据的安全。另外，函数返回其参数值似乎有些多余（事实上，在第 8 章中定义的 GetName（）函数就是无值型的），其实这样做完全是为了使用上的方便。经过这样的定义后，该函数就可以用在如下形式的语句中：

```
p = per.GetName(buff);
cout << per.GetName(buff) << endl;
```

其中：p 是一个字符型指针，buff 是一个足够大的字符型数组，而 per 则是一个已初始化了的 Person 类对象。

9.4 构造函数与对象成员

前边已经介绍过，一个类的对象可以作为另一个类的成员。当对一个包含有对象成员的类的对象进行初始化时，就需要调用其成员所属类的构造函数。

设有类说明：

```
class class_name {
        class_name1     c1;
        class_name2     c2;
        //...
        class_namen     cn;
        //...
};
```

则为了初始化对象成员，类 class_name 应有定义为如下形式的构造函数：

```
class_name :: class_name(args) : c1(args1), c2(args2),..., cn(argsn)
{
        //...
}
```

其中：$args_i$ 是为调用成员 c_i 的构造函数时所提供的参数；冒号“:”是C++语言的一个标点符号，其后跟着的用逗号分隔的列表叫做成员初始化列表。

例 9.13 初始化对象成员。

```
//      EX9_13.CPP

#include      <iostream.h>

class  C1     {
              int     x1;
       public:
              C1(int a)
              {
```

```
                x1 = a;
                cout << "Class C1 constructing..." << x1 << endl;
        }
        ~C1() {cout << "Class C1 destructing...\n";}
};

class   C2      {
        int     x2;
    public:
        C2(int a = 0)
        {
                x2 = a;
                cout << "Class C2 constructing..." << x2 << endl;
        }
        ~C2() {cout << "Class C2 destructing...\n";}
};

class   C3      {
        int     x3;
    public:
        C3(int a)
        {
                x3 = a;
                cout << "Class C3 constructing..." << x3 << endl;
        }
        ~C3() {cout << "Class C3 destructing...\n";}
};

class   C       {
        int     x;
        C1      c1;
        C2      c2;
        C3      c3;
    public:
        C(int, int, int);
        ~C()    {cout << "Class C destructing...\n";}
};

inline C :: C(int d, int a, int b) : c1(b), c3(d)
{
    x = a;
    cout << "Class C construsting..." << x << endl;
}

void    main()
{
    C       c(3, 1, 2);

    cout << "Now, object of class C is created.\n";
    cout << "Then, destroy it.\n";
}
```

该程序的输出为：

```
Class C1 construsting...2
Class C2 construsting...0
Class C3 construsting...3
Class C construsting...1
Now, object of class C is created.
Then, destroy it.
Class C destructing...
Class C3 destructing...
Class C2 destructing...
Class C1 destructing...
```

由程序的输出可以看出：当一个类包含有对象成员时，则在创建该类的对象时将按成员在类中说明的先后顺序调用对象成员的构造函数，最后才调用该对象的构造函数；而在撤消该对象时，则按相反的顺序调用对象成员的析构函数。

例 9.13 在类 C 中说明了一个初始化构造函数，该函数带有 3 个参数：其中第一个参数是用来初始化其对象成员 c3 的；第二个参数是用来初始化其数据成员 x 的；而第三个参数则是用来初始化对象成员 c1 的。由于对象成员 c2 带有缺省的初始化参数，本例中没有对它进行初始化，若需要用非缺省值对它进行初始化，可以在该构造函数中再增加一个参数。在定义该构造函数时，采用了上述的定义方式，即，在构造函数的参数表后跟一成员初始化列表，指出了哪个参数是初始化哪个对象成员的。构造函数参数表中参数说明的顺序并不要求与成员初始化列表所说明的顺序相一致，其对应关系由参数名指定。为说明问题，本例就是这样做的。在实用中，为了改善程序的可读性，建议按顺序列出参数和成员初始化列表。

应指出的是：带有成员初始化列表的形式仅用于构造函数的定义中，不得用在函数原型中，也不得用于类中的其它函数。只有在类中说明构造函数的同时定义它的函数体（即内联函数）时，才可以在类说明中使用该形式。另外，初始化列表不仅可以用来初始化类中的对象成员，也可以用来初始化类中的普通数据成员，如例 9.14 所示。

例 9.14 说明一个记录类，该记录类用来记载一个人的身分。

```
//      EX9_14.H

#if !defined   _EX9_14_H_
#define        _EX9_14_H_

#include       "ex9_12.h"

typedef        unsigned long UL;

class   Record{
        private:
                UL     ID_Num;
                Person Per;
                char   *Address;
        public:
                Record() : Per(), ID_Num(0), Address(0)    {}
```

```
            Record(UL, char*, int, char, char*);
            ~Record()     {delete Address;}
            void    Fill_In(UL, char*, int, char, char*);
            char    *GetAddr(char*);
            void    SetAddr(char *addr);
            void    Show();
};

#endif


//      EX9_14A.CPP

#include      <iostream.h>
#include      <string.h>
#include      "ex9_14.h"

Record :: Record(UL num, char * name, int age, char sex, char *addr)
                  : Per(name, age, sex), ID_Num(num)
{
      Address = new char[strlen(addr) + 1];
      strcpy(Address, addr);
}

void    Record :: Fill_In(UL num, char *name, int age, char sex, char *addr)
{
      ID_Num = num;
      Per.SetName(name);
      Per.SetAge(age);
      Per.SetSex(sex);
      if(Address)
            delete Address;
      Address = new char[strlen(addr) + 1];
      strcpy(Address, addr);
}

char    *Record :: GetAddr(char *str)
{
      strcpy(str, Address);

      return str;
}

void    Record :: SetAddr(char *addr)
{
      if(Address)
            delete Address;
      Address = new char[strlen(addr) + 1];
      strcpy(Address, addr);
}

void    Record :: Show()
```

```
{
        cout << "ID_No.: " << ID_Num << endl;
        Per.Show();
        cout << "Address: " << Address << endl;
}


//      EX9_14.CPP

#include      <iostream.h>
#include      <stdio.h>
#include      "ex9_14.h"

void    main()
{
        UL      n;
        int     ag;
        char    str1[20], str2[50], ch;
        Recordrec1(12345678, "Wang Ping", 23, 'f', "Jiefanglu road Xi'an");

        cout << "Please enter a person's record\n\tID_No.: ";
        cin >> n;
        cout << "\tName: ";
        gets(str1);
        cout << "\tAge: ";
        cin >> ag;
        cout << "\tSex(m/f): ";
        cin >> ch;
        cout << "\tAddress: ";
        gets(str2);

        Recordrec2;

        rec2.Fill_In(n, str1, ag, ch, str2);

        cout << "Person1:\n";
        rec1.Show();
        cout << "Person2:\n";
        rec2.Show();
}
```

在上例的源程序文件 EX9_14A.CPP 中，定义类 Record 缺省的构造函数时就是利用初始化列表将类中的 3 个数据成员均进行了初始化。

生成程序 EX9_14.EXE 时，可将文件 EX9_12.CPP，EX9_14A.CPP 和 EX9_14.CPP 分别编译后连接而成；亦可将这 3 个文件名放在一个工程文件中，一次编译连接而成。

运行该程序时，设用户对程序的响应为：

```
Please enter a person's record
        ID_No.: 23456789<CR>
        Name: Li Ming<CR>
        Age: 34<CR>
```

Sex(m/f): **m**<CR>
Address: **Changan road Peking**<CR>

则程序的输出为：

```
Person1:
ID_No.: 12345678
Name: Wang Ping Age: 23      Sex: f
Address: Jiefang road Xi'an
Person2:
ID_No.: 23456789
Name: Li Ming    Age: 34     Sex: m
Address: Changan road Peking
```

例 9.14 中用到了一个 C++库函数 gets（），该函数的原型在头文件 stdio.h 中说明为

```
char *gets(char *str);
```

其作用是请求从键盘上输入一个以回车结束的字符串，该函数接受所输入的串并把它存入参数 str 所指的缓冲区中（不包括回车符）。若函数操作成功，则返回指向 str 的指针；否则返回一个空指针。与 cin 相比，函数 gets（）允许输入串中包含有空白字符，因此，在要求输入象姓名、地址这样的包含有空白字符的串时，可以用该函数来代替 cin。

习题

9-1 为习题 8-2 增加适当的构造函数。

9-2 为习题 8-4，8-5 增加适当的构造函数。

9-3 将习题 7-3 所要求的结构数组改写成对象数组。

9-4 为以上 3 个习题中所说明的类定义析构函数。

9-5 为 9-4 中修改过的类定义拷贝初始化构造函数。

9-6 设有一个大小可变的整型数组类：

```
class Array {
    private:
        int    Size;              // 数组的大小
        int    *Arr;              // 数组的首地址
        int    ElemNum;           // 实际元素的个数
    public:
        Array(int sz = 100);      // 构造函数
        ~Array();                 // 析构函数
        int    &GetElem(int i);   // 获取下标为 i 的元素的引用
        int    GetElemNum(void);  // 获取实际元素的个数
};
```

试完成该类各成员函数的定义。注意，成员函数 GetElem（）可以用来为指定元素赋值。

第 10 章　继承和派生类

继承性是面向对象程序设计语言的基本特性之一。继承是指从已有类基础上建立新类的这一性质，它允许在构造软件系统的层次结构中可以利用已存在类的部分或全部内容，从而大大地提高了软件的重用性。

派生类则是继承的直接产物，它是通过继承已有的类而产生的新类，是代码重用的具体实现。

10.1　继承

继承这一概念最初是产生于生物学的分类概念。科学家对昆虫的分类与图 10-1 相似。在该图中，最高层最为一般化，每一层都比它的上一层更加具体化。一旦某个特征被定义下来，所有在它之下的种类都包含该特征，比如，一旦确定一个昆虫为苍蝇中的一个，则不必指出这个苍蝇有翅膀，该苍蝇所属的门类已从有翅门类中继承了这一特征。

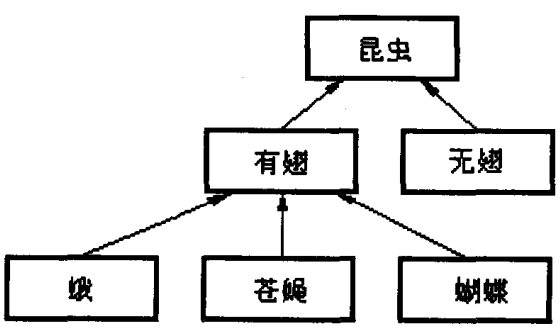

图 10-1　昆虫的分类树

在面向对象的程序设计语言中，继承类似于昆虫的分类关系，只是这里所说的继承是针对类类型而言的。在 C++语言中，称一个类继承另一个类的过程为派生一个类，所派生的类叫做派生类或子类，而被继承的类则叫做基类或父类。类的派生过程可以无限地进行下去，即派生类可以用作另一派生类的基类，这样建立起来的类体系称为类的等级结构，类的等级结构就类似于图 10-1 所示的分类树。

在 C++语言中，继承又分单一继承和多重继承。单一继承时，派生类只有一个基类，如图 10-2(a)所示；而多重继承时，派生类则可以有多个基类，如图 10-2(b)所示。

在图 10-2(a)中，学生和职工都是人，它们分别从人类中继承了人所具有的共性并增加了各自的特性，比如，学生的专业、所在班级；职工的工种、所属车间。在图 10-2(b)中，职工大学生是正在学校学习的职工，他们既具有职工的特点又具有学生的特点，因此，职工大

学生类就从学生类和职工类中综合派生出来。

图 10-2　单一继承和多重继承

10.1.1　单一继承

通过单一继承派生一个类的一般形式为：

```
class derivative : <access> base
{
    <private:>
        members;
    public:
        members;
    <protected:
        members;>
};
```

其中：derivative 为派生类的类名；base 为该类所继承的类即基类的类名；而 access 为访问控制字，它必须是关键字 private 或 public 之一。

当访问控制字为 private 时，称类 derivative 从类 base 私有派生；而当访问控制字为 public 时，称类 derivative 从类 base 公有派生。访问控制字可以省略，这时，对类而言，缺省为私有派生；而对结构而言则缺省为公有派生。

10.1.1.1　公有派生

公有派生时，基类成员的访问权限在派生类中保持不变，即原来在基类中定义为私有的成员在派生类中仍为私有的；原来在基类中定义为公有的成员在派生类中仍为公有的。这就意味着在类外可以访问派生类中从基类继承下来的公有成员。

例10.1　公有派生。

```
//    EX10_1.H

class  B     {
             int     x, y;
       public:
             void    SetX(int a)    {x = a;}
             void    SetY(int b)    {y = b;}
             int     GetX() {return x;}
```

```
        int     GetY() {return y;}
};


//      EX10_1.CPP

#include      <iostream.h>
#include      "ex10_1.h"

class   D : public B   {
        int     z;
    public:
        void    SetZ(int c)     {z = c;}
        int     GetZ() {return z;}
        int     Sum()  {return (z + GetX() + GetY());}
};

void    main()
{
    D     d;

    d.SetX(1);
    d.SetY(2);
    d.SetZ(3);
    cout << "X = " << d.GetX() << '\t' << "Y = " << d.GetY() << '\t' << "Z = " << d.GetZ() << endl;
    cout << "Sum = " << d.Sum() << endl;
}
```

该程序的输出为：

```
X = 1       Y = 2       Z = 3
Sum = 6
```

从例 10.1 的输出可以看出，当一个类从另一个类公有派生时，基类中的公有成员函数就如同是在派生类中定义的一样，在类外可以通过派生类的对象名来直接访问。然而，对基类而言，派生类也属于它的“外部”，因此，在派生类中也不能访问基类中的私有成员。例如，类 D 的成员函数 Sum（）的定义就不能写成：

```
void  D :: Sum()
{
    return (z + x + y);
}
```

在派生类中也必须通过基类所提供的公共接口才可以访问基类中的私有成员。

10.1.1.2 私有派生

私有派生时，基类中所有成员的访问权限在派生类中均为私有的。即从派生类外部来看，其基类的所有成员均不可见。

例10.2 私有派生。

```
//      EX10_2.CPP

#include      <iostream.h>
#include      "ex10_1.h"

class   D : private B  {
              int      z;
        public:
              void     SetZ(int c)     {z = c;}
              void     SetBase(int a, int b)
              {
                       SetX(a);
                       SetY(b);
              }
              int      GetBX()         {return GetX();}
              int      GetBY()         {return GetY();}
              int      GetZ() {return z;}
              int      Sum()  {return (z + GetX() + GetY());}
};

void    main()
{
        D       d;

        d.SetBase(1, 2);
        d.SetZ(3);
        cout << "X = " << d.GetBX() << '\t' << "Y = " << d.GetBY() << '\t' << "Z = " << d.GetZ() << endl;
        cout << "Sum = " << d.Sum() << endl;
}
```

该程序的输出与例 10.1 相同。

由于在例 10.2 中类 D 是从类 B 私有派生而来的，这时，从类体系的角度来看，基类 B 中公有成员所提供的公共接口则只是相对其派生类 D 而言的，这些成员是派生类以外的程序所不可见的。因此，为了能向基类中的数据成员置数以及取得基类中数据成员的值，在派生类中必须定义相应的公有成员函数。请读者仔细对比例 10.1 和例 10.2，体会公有派生与私有派生的区别。

10.1.1.3 保护成员

在类定义中，说明为 protected 的成员称做保护成员。保护成员具有双重作用：对于其派生类而言，它是公有的；而对于其外部的程序而言，它则是私有的。

例10.3 保护成员。

```
//      EX10_3.H

class   B       {
        protected:
                int   x, y;
        public:
                void SetX(int a)       {x = a;}
```

```
        void SetY(int b)      {y = b;}
        int   GetX()      {return x;}
        int   GetY()      {return y;}
};

// EX10_3.CPP

#include      <iostream.h>
#include      "ex10_3.h"

class   D : public B    {
            int     z;
      public:
            void    SetZ(int c)     {z = c;}
            void    SetBase(int a, int b)
            {
                  x = a;
                  y = b;
            }
            int     GetBX()       {return x;}
            int     GetBY()       {return y;}
            int     GetZ() {return z;}
            int     Sum()  {return (z + x + y);}
};

void    main()
{
      D     d;

      d.SetBase(1, 2);
      d.SetZ(3);
      cout << "X = " << d.GetBX() << '\t' << "Y = " << d.GetBY() << '\t' << "Z = " << d.GetZ() << endl;
      cout << "Sum = " << d.Sum() << endl;
}
```

该程序的输出与例 10.1 相同。

在例 10.3 中，基类 B 的数据成员被说明成保护成员。对其派生类而言，这些数据成员就如同在基类中被说明成公有的一样，因而可以直接对它们进行访问，提高了程序运行效率。

通常，若一个类主要是作为基类以供派生新类而用，其数据成员说明成保护的比较方便。但在这种情况下，若由于某种原因而改变了保护成员的表示形式，例如，将类 B 中的成员 x 改为数组，则这些改变也要影响到派生类。因此，在实用中应仔细权衡程序的效率与程序的可维护性，以决定是否采用保护成员。

就例 10.3 而言，类 D 无论是从类 B 中公有派生还是私有派生均是没有区别的。然而，对于类 D 的派生类而言，类 D 的派生方式则会对它们产生不同的影响。若类 D 是从类 B 公有派生的，则类 B 中的数据成员 x 和 y 在类 D 中仍是保护成员，因而类 D 的派生类也可以直接访问类 B 中的这两个成员；若类 D 从类 B 私有派生，则由于其基类所有成员在类 D 中都是私有的，因而使得类 D 的派生类不能直接访问类 B 中的保护成员。

在类体系中，访问权限的继承关系如表 10.1 所示。

表 10.1　类体系中访问权限的继承关系

派生性质	基类中的访问权限	派生类中的访问权限
public	public	public
	protected	protected
	private	private
private	public	private
	protected	private
	private	private

10.1.1.4　抽象类与保护的成员函数

所谓抽象类是指只能作为基类来派生新类的类类型，在程序中不能说明抽象类的对象。

有多种因素可以使得一个类成为抽象类，保护的构造函数就是这样的一个因素。当一个类中包含有保护的构造函数时，因为这样的构造函数对除该类之派生类以外的所有外部程序来讲是私有的，所以，外部程序由于无法调用该构造函数而不能创建该类的对象。对该类的派生类来讲，该构造函数却是公有的，因而在创建其派生类的对象时就可以调用它为基类成员分配内存。

保护的析构函数同样阻止了在撤消对象时对它的调用，因此，若一个类的析构函数被说明为保护的，则该类也是一个抽象类。

10.1.2　多重继承

通过多重继承派生一个类的一般形式为：

```
class derivative : <access> base1, <access> base2, ..., <access> baseN
{
        <private:>
                members;
        public:
                members;
        <protected:
                members;>
};
```

可见，多重继承可视为单一继承的扩展。

例 10.4　多重继承。

```
//      EX10_4A.H

class   A       {
        protected:
                int     w;
        public:
                void    SetW(int a)     {w = a;}
                int     GetW()          {return w;}
```

```
};

// EX10_4.H

#include      "ex10_4a.h"
#include      "ex10_3.h"

class   D : public A, B       {              // A
              int     z;
        public:
              void    SetZ(int a)     {z = a;}
              void    SetBase(int a, int b, int c)
              {
                    x = a;
                    y = b;
                    w = c;
              }
              int     GetBX()       {return x;}
              int     GetBY()       {return y;}
              int     GetBW()       {return w;}
              int     GetZ() {return z;}
              int     Sum()  {return (z + x + y + w);}
};

//    EX10_4.CPP

#include      <iostream.h>
#include      "ex10_4.h"

void    main()
{
        D       d;

        d.SetBase(1, 2, 3);
        d.SetZ(4);
        cout << "X = " << d.GetBX() << '\t' << "Y = " << d.GetBY() << '\t'
              << "W = " << d.GetBW() << '\t' << "Z = " << d.GetZ() << endl;
        cout << "Sum = " << d.Sum() << endl;
}
```

该程序的输出为：

```
X = 1       Y = 2       W = 3       Z = 4
Sum = 10
```

注意：行 A 中说明派生类 D 时，其基类 B 前边没有访问控制字，这里实际上使用了 C++ 语言的缺省访问控制，即 private。不要将这一说明理解成基类 B 与基类 A 共用一个访问控制字 public。

10.2 初始化基类成员

当派生类和基类中都定义有初始化构造函数时，则可以在创建派生类的对象时调用基类中相应的构造函数来初始化基类中的成员。带有初始化基类成员的派生类初始化构造函数的定义具有如下的一般形式：

```
derivative :: derivative : base1(args), base2(args),..., basen(args)
{
    //...
}
```

例 10.5 初始化基类成员。

```
//      EX10_5.H

class   B       {
        protected:
                int     x, y;
        public:
                B(int a, int b) : x(a), y(b)      {}
                void    SetX(int a)     {x = a;}
                void    SetY(int b)     {y = b;}
                int     GetX() {return x;}
                int     GetY() {return y;}
};

class   A       {
        protected:
                int     w;
        public:
                A(int a) : w(a) {}
                void    SetW(int a)     {w = a;}
                int     GetW()          {return w;}
};

class   D : public A, private B         {
        private:
                int     z;
        public:
                D(int a, int b, int c, int d) : B(a, b), A(c), z(d)         {}
                void    SetZ(int a)     {z = a;}
                void    SetBase(int a, int b, int c)
                {
                        x = a;
                        y = b;
                        z = c;
                }
                int     GetBX()         {return x;}
                int     GetBY()         {return y;}
```

```
        int     GetBW()     {return w;}
        int     GetZ() {return z;}
        int     Sum()  {return (z + x + y + w);}
};


//      EX10_5.CPP

#include    <iostream.h>
#include    "ex10_5.h"

void    main()
{
    D     d(1, 2, 3, 4)

    cout << "X = " << d.GetBX() << '\t' << "Y = " << d.GetBY() << '\t'
        << "W = " << d.GetBW() << '\t' << "Z = " << d.GetZ() << endl;
    cout << "Sum = " << d.Sum() << endl;
}
```

该程序的输出同例 10.4。

下面给出几个实用的例子。

例10.6 说明一个学生类。

```
//      EX10_6.H

#if !defined    _EX10_6_H_
#define     _EX10_6_H_

#include    "ex9_12.h"

typedef     unsigned long UL;

class   Student : public Person     {
    private:
        char    *Department;                    // 系部
        char    *Speciality;                    // 专业
        UL      Number;                         // 学号
        void    CopyStrings(char*, char*);
    public:
        Student() : Person(), Department(0), Speciality(0), Number(0)    {}
        Student(char*, int, char, char*, char*, UL);
        Student(Student&);
        ~Student()
        {
            delete Department;
            delete Speciality;
        }
        void    SetDep(char*);
        void    SetSpec(char*);
```

```
        void    SetNum(UL num)          {Number = num;}
        char    *GetDep(char*);
        char    *GetSpec(char*);
        UL      GetNum()        {return Number;}
};

#endif


//      EX10_6A.CPP

#include      <string.h>
#include      "ex10_6.h"

void    Student :: CopyStrings(char *sP, char * sS)
{
        Department = new char[strlen(sP) + 1];
        strcpy(Department, sP);
        Speciality = new char[strlen(sS) + 1];
        strcpy(Speciality, sS);
}

Student :: Student(char *name, int age, char sex, char *dep, char *spec, UL num)
                : Person(name, age, sex), Number(num)
{
        CopyStrings(dep, spec);
}

Student :: Student(Student &stu) : Person(stu), Number(stu.Number)
{
        CopyStrings(stu.Department, stu.Speciality);
}

void    Student :: SetDep(char *dep)
{
        if(Department)
                delete Department;
        Department = new char[strlen(dep) + 1];
        strcpy(Department, dep);
}

void    Student :: SetSpec(char *spec)
{
        if(Speciality)
                delete Speciality;
        Speciality = new char[strlen(spec) + 1];
        strcpy(Speciality, spec);
}

char    *Student :: GetDep(char *str)
{
```

```
        strcpy(str, Department);

        return str;
}

char    *Student :: GetSpec(char *str)
{
        strcpy(str, Speciality);

        return str;
}


//      EX10_6.CPP

#include      <iostream.h>
#include      <stdio.h>
#include      "ex10_6.h"

void    main()
{
        Student       stu1("Wang Ping", 23, 'f', "Computer Dep.",
                      "Computer Software", 1234567), stu2(stu1), stu3;
        char    str[81], ch;
        int     ag;
        UL      num;

        cout << "Please enter a student's data:\n\tName: ";
        gets(str);
        stu3.SetName(str);
        cout << "\tAge: ";
        cin >> ag;
        stu3.SetAge(ag);
        cout << "\tSex(m/f): ";
        cin >> ch;
        stu3.SetSex(ch);
        cout << "\tDepartment: ";
        gets(str);
        stu3.SetDep(str);
        cout << "\tSpeciality: ";
        gets(str);
        stu3.SetSpec(str);
        cout <<"\tStudent Code: ";
        cin >> num;
        stu3.SetNum(num);

        cout << "\nStudent1:\n";
        cout << "Nmae: " << stu1.GetName(str) << '\t' << "Age: "
             << stu1.GetAge() << '\t' << "Sex: "
             << stu1.GetSex() <<endl;
        cout << "Dep.: " << stu1.GetDep(str) << '\t' << "Spec.: "
```

```
            << stu1.GetSpec(str) << '\t' << "Code: "
            << stu1.GetNum() << endl;

    cout << "Student2:\n";
    cout << "Nmae: " << stu2.GetName(str) << '\t' << "Age: "
            << stu2.GetAge() << '\t' << "Sex: "
            << stu2.GetSex() << endl;
    cout << "Dep.: " << stu2.GetDep(str) << '\t' << "Spec.: "
            << stu2.GetSpec(str) << '\t' << "Code: "
            << stu2.GetNum() << endl;

    cout << "Student3:\n";
    cout << "Nmae: " << stu3.GetName(str) << '\t' << "Age: "
            << stu3.GetAge() << '\t' << "Sex: "
            << stu3.GetSex() << endl;
    cout << "Dep.: " << stu3.GetDep(str) << '\t' << "Spec.: "
            << stu3.GetSpec(str) << '\t' << "Code: "
            << stu3.GetNum() << endl;
}
```

将例 10.5 生成可执行程序时，应连接 EX10_6.CPP，EX9_12.CPP 和 EX10_6A.CPP。

注意文件 EX10_6A.CPP 中拷贝初始化构造函数 Student :: Student（Student&)。在该函数中，使用了类 Student 的对象来初始化其基类 Perosn 中的数据成员。这种初始化基类成员的方式是合法的，这时，编译器将用派生类对象 stu 中属于基类成员的值来初始化基类中的相应成员。

例10.7 定义一个职工类。

```
//      EX10_7.H

#if !defined   _EX10_7_H_
#define        _EX10_7_H_

#include       "ex9_12.h"

class   Worker : public Person      {
        private:
                char    *Department;        // 部门
                char    *Job;               // 职务
        public:
                Worker() : Person(), Department(0), Jub(0) {}
                Worker(char*, int, char, char*, char*);
                Worker(Worker&);
                ~Worker()
                {
                        delete Department;
                        delete Job;
                }
                void    SetDep(char*);
                void    SetJob(char*);
                char    *GetDep(char*);
```

```
        char    *GetJob(char*);
};

#endif


//      EX10_7.CPP

#include      <string.h>
#include      "ex10_7.h"

Worker :: Worker(char *name, int age, char sex, char * dep, char *job) : Person(name, age, sex)
{
      Department = new char[strlen(dep) + 1];
      strcpy(Department, dep);
      Job = new char[strlen(job) + 1];
      strcpy(Job, job);
}

Worker :: Worker(Worker *work) : Person(work)
{
      Department = new char[strlen(work.Department) + 1];
      strcpy(Department, work.Department);
      Job = new char[strlen(work.Job) + 1];
      strcpy(Job, work.Job);
}

void   Worker :: SetDep(char *dep)
{
      if(Department)
            delete Department;
      Department = new char[strlen(dep) + 1];
      strcpy(Department, dep);
}

void   Worker :: SetJob(char *job)
{
      if(Job)
            delete Job;
      Job = new char[strlen(job) + 1];
      strcpy(Job, job);
}

char   *Worker :: GetDep(char *dep)
{
      strcpy(Department, dep);

      return dep;
}

char   *Worker :: GetJob(char *jov)
```

```
{
        strcpy(Job, job);

        return job;
}
```

例 10.7 给出了职工类 Worker 的定义，读者可仿照例 10.6 自行设计测试该类的程序。

10.3 二义性、支配规则和赋值兼容规则

在类体系中，基类中成员的重名会产生二义性；而派生类中的成员将支配其基类中的同名成员。

10.3.1 二义性

由于继承与派生过程中会不断地说明新的成员以及多重继承时派生类中拥有多个基类，因而难免会出现成员重名的现象。如果在一个表达式中所访问的成员涉及到类体系中不止一个的成员，则这种访问就是有二义性的。

例 10.8 有二义性的例子。

```
//      EX10_8.H

#include      <iostream.h>

class   B1    {
        protected:
                int     x, y;
        public:
                void    Show()
                {
                        cout << "B1.X= " << x << '\t' << "B1.Y = " << y << endl;
                }
};

class   B2    {
        protected:
                int     x;
        public:
                void    Show()
                {
                        cout << "B2.X = " << x << endl;
                }
};


//      EX10_8.CPP

#include      "ex10_8.h"
```

```
class   D : public B1, public B2      {
        public:
                void    SetX(int a)     {x = a;}        // A
};

void    main()
{
        D       d;

        d.Show();                                       // B
}
```

该程序在编译过程中，将会产生两个编译错误：一个在行 A，编译器不清楚这里的 x 是指基类 B1 中的成员 x 还是基类 B2 中的成员 x；一个在行 B，同样，编译器不清楚这里的函数 Show（）是指基类 B1 中的成员函数还是基类 B2 中的成员函数。这两个错误就是由于二义性而引起的。

使用成员名限定可以消除二义性。

例10.8 使用成员名限定消除二义性。

```
//      EX10_9.CPP

#include      "ex10_8.h"

class   D : public B1, public B2      {
        public:
                void    SetX(int a)     {B1 :: x = a;}          // A
};

void    main()
{
        D       d;

        d.B2 :: Show();                                         // B
}
```

该例在行 A 中使用成员名 B1 与作用域运算符“::”，明确地指出函数中是对基类 B1 中的成员 x 进行赋值；行 B 以同样的方法指明调用基类 B2 中的成员函数 Show（）。注意：在头文件 EX10_8.H 中，类 B1 和 B2 中的数据成员均被说明成保护成员，若类 B1 中的 x 是一个私有成员，则行 A 的赋值操作即为非法的，这时，类 B1 中必须提供公有的为数据成员赋值的方法。

在多重继承时，如果派生类的多个基类拥有一个共同的基类，则在派生类中访问该共同基类中的成员时就存在二义性。

例10.9 定义一个职工大学学员类。

```
//      EX10_10.H

#include      "ex10_6.h"
#include      "ex10_7.h"
```

```
class   Work_Stu : public Worker, public Student  {
        public:
                Work_Stu();
                Work_Stu(Work_Stu &ws) : Worker(ws), Student(ws)     {}
                Work_Stu(char *name, int age, char sex, char *depw, char *job, char *deps, char *spec, UL num)
                        : Worker(name, age, sex, depw, job), Student(name, age, sex, deps, spec, num)  {}
};


//      EX10_10.CPP

#include      <iostream.h>
#include      "ex10_10.h"

void    main()
{
        Work_Stu      ws("Wang Ping", 23, 'f', "Manager office", "Secretary",
                      "Computer Dep.", "Computer application", 1234567);

        cout << "Name: " << ws.Student::GetName() << '\t'
                << "Age: " << ws.Worker::GetAge() << '\t'
                << "Sex: " << ws.Worker::GetSex() << endl;
        cout << "Work Dep.: " << ws.Worker::GetDep() << '\t'
                << "Job: " << ws.GetJob() << endl;
        cout << "Collage Dep.: " << ws.Student::GetDep() << '\t'
                << "Speciality: " << ws.GetSpec() << '\t'
                << "Student code: " << ws.GetNum() << endl;
}
```

请读者自行分析主函数中访问对象 ws 中各成员时的路径。

10.3.2 支配规则

支配规则是指派生类中的成员支配其基类中的同名成员。当一个名字支配另一个名字时，不存在二义性，对该名字的访问是对支配者的访问。

例10.10 支配规则。

```
//      EX10_11.CPP

#include      "ex10_8.h"

class   D : public B1, public B2     {
                int     x;
        public:
                void    SetX(int a)     {x = a;}                              // A
                void    Show()          {cout << "D.X = " << x << endl;}      // B
};

void    main()
{
        D       d;
```

```
    d.Show();                                    // C
}
```

该程序不存在二义性，其中行 A 与行 B 中对 x 的访问均是对派生类 D 中成员 x 的访问，而行 C 中的函数调用也是调用派生类 C 中的成员函数 Show（）。这是由于支配规则在起作用之故。

10.3.3 继承和对象成员

由于存在二义性，所以一个类不得从同一个类中直接继承一次以上。例如，欲定义一个 3 口之家的家庭类：

```
class  Family : private Person, private Person, private Person  {
    //...
};
```

这时将因为二义性使得类 Family 中所有属于 Person 类的成员根本无法访问。类似这种情况，应在类 Family 中使用 Person 类的对象作为成员为宜：

```
class  Family     {
            Person      Father, Mother, Child;
    //...
};
```

实际上，在一个家庭中，人是家庭的成员，而并非家庭是一个人，因此使用 Person 类的对象作为类 Family 的成员在逻辑上也更为合理。

从程序运行的效率来看，使用继承和使用对象成员是没有什么差别的。在实用中应从类的逻辑关系着眼，正确地选用继承或对象成员。

10.3.4 赋值兼容规则

赋值兼容规则是指在公有派生情况下，一个派生类的对象可以用在其基类对象可使用的地方。设：

```
class  B     {//...};
class  D : public B {//...};
```

则：

1. 派生类的对象可以向基类的对象赋值。例如：

```
D     d;
B     b;

b = d;
```

这时，将对象 d 中所含的 B 类成员的值赋给了对象 b。

2. 派生类的对象可以初始化基类的引用。例如：

```
D    d;
B    &rb = d;
```

3. 可以将派生类对象的地址赋指向基类类型的指针。例如：

```
D    d;
B    *pb = &d;
```

注意：在 2，3 两种情况下，利用指针 pb 和引用 rb 只能访问派生类对象 d 从基类 B 中继承下来的成员。若欲利用它们访问类 D 中新增的成员，则须对它们进行强制类型转换。比如，利用 pb 指针访问对象 d 中的某个公有成员函数 fun（）：

```
(D*)pb->fun();
```

10.4 虚基类

例 10.10 中类 Work_Stu 的派生关系如图 10-3(a)所示，该类的一个对象中数据成员在内存中的存储形式如图 10-3(b)所示。

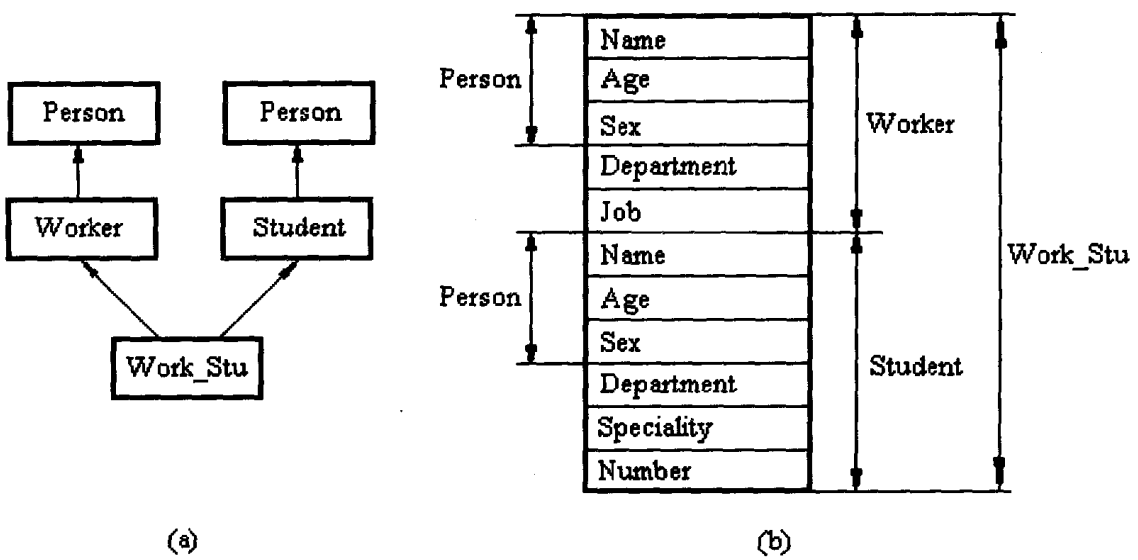

图 10-3 类体系 Work_Stu 及其对象中数据成员的存储形式

由图 10-3 可见，在类 Work_Stu 的对象中存在有两个类 Person 的实例。实际上，在 C++ 语言中，如果在多条继承路径上有一个公共的基类，则在这些路径中某几条路径的汇合处将会产生该公共基类的多个拷贝。这一情况不仅造成了存储单元的浪费，而且还产生了二义性。

若欲使某公共基类只在派生类中生成一个拷贝，则应将该基类说明为虚基类。说明虚基类的一般形式为：

```
class derivative : virtual <access> base        {
    //...
};
```

其中，关键字 virtual 可以放在访问控制字之前，也可以放在访问控制字之后。

例10.11 职大学员类的改进。

```
//      EX10_12A.H

#if !defined   _EX10_12A_H_
#define        _EX10_12A_H_

#include       "ex9_12.h"

typedef        unsigned long UL;

class  Student : virtual public Person       {
       private:
              char    *Department;
              char    *Speciality;
              UL      Number;
              void    Init(char*, char*, UL);
       public:
              Student()       {Init(0, 0, 0L);}
              Student(char*, int, char, char*, char*, UL);
              Student(Student &s)  {Init(s.Department, s.Speciality, s.Number);}
              Student(char *d, char *s, UL n)       {Init(d, s, n);}
              ~Student()
              {
                     delete Department;
                     delete Speciality;
              }
              void    SetDep(char*);
              void    SetSpec(char*);
              void    SetNum(UL num)     {Number = num;}
              char    *GetDep(char*);
              char    *GetSpec(char*);
              UL      GetNum()      {return Number;}
};

#endif


//      EX10_12A.CPP

#include       <string.h>
#include       "ex10_12a.h"

void   Student :: Init(char *dep, char *spec, UL num)
{
       Department = new char[strlen(dep) + 1];
       strcpy(Department, dep);
       Speciality = new char[strlen(spec) + 1];
       strcpy(Speciality, spec);
       Number = num;
```

```
}

Student :: Student(char *name, int age, char sex, char *dep, char *spec, UL num) : Person(name, age, sex)
{
        Init(dep, spec, num);
}

void    Student :: SetDep(char *dep)
{
        if(Department)
                delete Department;
        Department = new char[strlen(dep) + 1];
        strcpy(Department, dep);
}

void    Student :: SetSpec(char *spec)
{
        if(Speciality)
                delete Speciality;
        Speciality = new char[strlen(spec) + 1];
        strcpy(Speciality, spec);
}

char    *Student :: GetDep(char *buff)
{
        strcpy(buff, Department);

        return buff;
}

char    *Student :: GetSpec(char *buff)
{
        strcpy(buff, Speciality);

        return buff;
}


//      EX10_12B.H

#if !defined    _EX10_12B_H_
#define         _EX10_12B_H_

#include        "ex9_12.h"

class   Worker : virtual public Person        {
        private:
                char    *Department;
                char    *Job;
                void    Init(char*, char*);
        public:
```

```
        Worker()        {Init(0, 0);}
        Worker(char*, int, char, char*, char*);
        Worker(Worker &w) {Init(w.Department, w.Job);}
        Worker(char *d, char *j)        {Init(d, j);}
        ~Worker()
        {
                delete Department;
                delete Job;
        }
        void    SetDep(char*);
        void    SetJob(char*);
        char    *GetDep(char*);
        char    *GetJob(char*);
};

#endif


//      EX10_12B.CPP

#include      <string.h>
#include      "ex10_12b.h"

void    Worker :: Init(char *dep, char *job)
{
        Department = new char[strlen(dep) + 1];
        strcpy(Department, dep);
        Job = new char[strlen(job) + 1];
        strcpy(Job, job);
}

Worker :: Worker(char *name, int age, char sex, char *dep, char *job) : Person(name, age, sex)
{
        Init(dep, job);
}

void Worker :: SetDep(char *dep)
{
        if(Department)
                delete Department;
        Department = new char[strlen(dep) + 1];
        strcpy(Department, dep);
}

void    Worker :: SetJob(char *job)
{
        if(Job)
                delete Job;
        Job = new char[strlen(job) + 1];
        strcpy(Job, job);
}
```

```
char    *Worker :: GetDep(char *buff)
{
        strcpy(buff, Department);

        return buff;
}

char    *Worker :: GetJob(char *buff)
{
        strcpy(buff, Jub);

        return buff;
}


//      EX10_12.H

#include      "ex10_12a.h"
#include      "ex10_12b.h"

class   Work_Stu : public Worker, public Student  {
        public:
                Work_Stu();
                Work_Stu(Work_Stu &ws) : Worker(ws), Student(ws)      { }
                Work_Stu(char *name, int age, char sex, char *depw, char *job, char *deps, char *spec, UL num)
                        : Person(name, age, sex), Worker(depw, job), Student(deps, spec, num)  { }
};


//      EX10_12.CPP

#include      <iostream.h>
#include      "ex10_12.h"

void    main()
{
        Work_Stu      ws("Wang Ping", 23, 'm', "Manager office", "Secretary",
                      "Computer Dep.", "Computer application", 1234567);
        char    str[81];

        cout << "Name: " << ws.GetName(str) << '\t'
                << "Age: " << ws.GetAge() << '\t'
                << "Sex: " << ws.GetSex() << endl;
        cout << "Work Dep.: " << ws.Worker::GetDep(str) << '\t';          // A
        cout << "Job: " << ws.GetJob(str) << endl;
        cout << "College Dep.: " << ws.Student::GetDep(str) << '\t';      // B
        cout << "Speciality: " << ws.GetSpec(str) << endl;
        cout << "Student Code: " << ws.GetNum() << endl;
}
```

这里，类 Student 和 Worker 将类 Person 作为它们的虚基类，从而使得在它们的派生类 Work_Stu 中仅保存一个 Person 类的拷贝。改进后的类体系 Work_Stu 及其对象中数据成员的存储形式如图 10-4 所示。

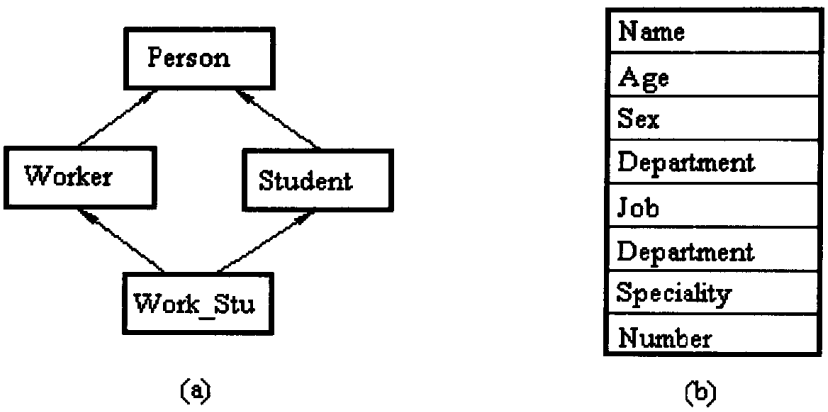

图 10-4　改进后的类体系 Work_Stu 及其对象中数据成员的存储形式

由于类 Person 是一个虚基类，因而在创建类 Work_Stu 的对象时，若调用类 Student 的构造函数：

```
Student :: Student(char*, int, char, char*, char*, UL);
```

和类 Worker 的构造函数：

```
Worker :: Worker(char*, int, char, char*, char*);
```

将不对 Person 的成员进行初始化（尽管这两个函数的定义中均包含有对 Person 成员初始化的操作），这是因为编译器不知道该通过哪个函数来初始化这唯一的基类。因此，本例就将对类 Person 成员的初始化工作交给类 Work_Stu 的构造函数来完成，这一点对比 EX10_10.H 与 EX10_12.H 中构造函数：

```
Work_Stu :: Work_Stu(char*, int, char, char*, char*, char*, char*, UL)
```

就可以看出。为此，又在类 Student 和 Worker 中分别增加了构造函数：

```
Student :: Student(char*, char*, UL);
```

和

```
Worker :: Worker(char*, char*);
```

程序中行 A 与行 B 中使用了成员名限定以消除二义性。

若在类 Work_Stu 增加两个公有成员函数：

```
char  *Work_Stu :: GetDepS()      {return Student :: GetDep();}
char  *Work_Stu :: GetDepW()      {return Worker :: GetDep();}
```

则使用起来将显得更加自然，这一改动作为练习请读者来完成。

应当说明的是：若某个类的基类是一个虚基类，则该类对象的建立与使用与普通类对象

的建立与使用一样。比如，用文件 EX10_12A.H 与 EX10_12A.CPP 的内容分别替换文件 EX10_6.H 和 EX10_6A.CPP，则例 10.6 中程序的功能不变。因此，也可以将类 Work_Stu 的基类说明成虚基类：

```
class    Work_Stu : virtual public Worker, virtual public Student  {
    //...
};
```

以便进一步派生时将 Work_Stu 用作基类。

习题

10-1 从类 Person 中派生出一个教师类，新增的属性有：专业、职称和主讲课程（一门），并为这些属性定义相应的方法。

10-2 许多研究生即具有学生的属性，又具有教师的属性。试通过多重继承说明一个研究生类。

10-3 从习题 9-7 中定义的类 Array 中派生出一个类，当建立该类的对象时允许程序指定数组的下标取值范围。其缺省的下标下界为 1；上界为 100。该类应能对下标超界进行检测。

10-4 什么是支配规则？设有以下的类：

```
class a     {                          class B : A {
            int    a;                              int    b;
    public:                            public:
            void   func();                         void   func();
    //...                                          //...
};                                     };
```

并设 b 为类 B 的对象，则语句 b.func();访问的是哪一个函数？

10-5 设有类：

```
class X     {                          class Y     {
    public:                                public:
            void   func();                         void   func();
    //...                                  //...
};                                     };
class Z : public X, public Y    {//...};
```

并设 z 为类 Z 的对象，则语句 z.func();是否存在二义性？若是，如何消除？

10-6 习题 10-2 的研究生类中可能存在两个 Person 类成员的拷贝。试对该类做适当的修改，使其基类成员不至于重复。

10-7 编写程序对例 10.7 中的类 Worker 进行测试。

10-8 修改例 10.12 中的 Work_Stu 类，使其更加合理、自然。

第 11 章　类的其它特性

本章介绍友元函数、虚函数、静态成员、const 对象和 volatile 对象，以及指向类成员的指针。

11.1　友元函数

虽然封装机制所带来的好处是极其明显的，然而，若绝对不允许类外的函数访问类中私有成员的话，确实也有许多不便之处。C++语言中的友元函数可以为在类外访问类中私有成员和保护成员提供方便。

11.1.1　友元函数的说明

一个类的友元函数（简称友元）是在该类中说明的、用关键字 friend 修饰的函数，该函数有权访问类中所有的成员。说明一个友元的一般形式为

friend type func_name(<arg_list>);

例 11.1　友元函数。

```
//      EX11_1.H

class  C      {
              int      x;
       public:
              C(int a)        {x = a;}
              int      GetX() {return x;}
              friend long     Cube(C&);                    // A
};

long   Cube(C &c)
{
       return(c.x * c.x * c.x);
}


// EX11_1.CPP

#include      <iostream.h>
#include      "ex11_1.h"

void   main()
{
```

```
    C     c(5);
    cout << Cube(c) << endl;                    // B
}
```

该程序的输出为

125

注意：友元函数 Cube（）带有一个参数——对类 C 的引用，并在函数的定义中利用该参数来访问对象 c 中的私有成员 x。这里必须这样做，这是因为友元虽在类中说明，但它并不是类的成员函数，因此它不带有 this 指针之故。

11.1.2 使用友元函数

友元实际上就是一个一般的函数，与其他普通函数不同之点在于：友元须在某个类中说明，它拥有访问说明它的类中所有成员的特权。而其他普通函数只能访问类中的公有成员。

虽然友元是在类中说明的，但它的作用域却在类外。友元作用域起始于其说明处，结束点与类名相同。例如，例 11.1 中的类 C 具有全局作用域，因而在该类中说明的友元 Cube（）亦具有全局作用域，这就是说在程序中的任何地方都可以调用该函数。

使用友元主要目的是提高程序运行效率。以例 11.1 为例，若没有友元，则主函数中的行 B 就得改为

```
cout << long(c.GetX() * c.GetX() * c.Getx()) << endl;
```

这需要 3 次调用类 C 的成员函数 GetX（）。而使用了友元后，则只需一次函数调用。

若一个友元在类中说明的同时进行了定义，则该函数就是内联的。例如，可以将例 11.1 中的行 A 改写为

```
friend long  Cube(&c)   {return c.x * c.x * c.x;}
```

就将友元函数 Cube（）说明为内联的。当然也可以在类外用关键字 inline 将友元说明为内联函数。

友元说明可以出现在类的私有部分、保护部分和公有部分，但这没有什么区别。在某类中说明友元只是说明该类允许这个函数随意访问它的所有成员，该函数并不是它的成员函数，指定对该函数的访问权限是没有意义的。

应强调说明的是：使用友元虽然可以提高程序的运行效率，但却破坏了类的封装性。友元就如同在封装类中成员的容器上打了一些洞，一个类拥有的友元越多，这种洞就越多，类的封装性就越差。因此，在实际应用中应慎重地使用友元。

11.1.3 将成员函数用作友元

可以将一个类的成员函数（包括构造函数与析构函数）说明为另一个类的友元。

例 11.2 将成员函数作为友元。

```
//      EX11_2.CPP
```

```
#include    <iostream.h>

class  Y;                                    // 超前说明

class  X    {
    private:
        int     x;
    public:
        X(int a = 0)   {x = a;}
        int     GetX() {return x;}
        void    SetX(Y&);
};

class  Y    {
    private:
        int     y;
    public:
        Y(int b = 0)   {y = b;}
        int     GetY() {return y;}
        friend  void X :: SetX(Y&);
};

void   X :: SetX(Y &r)
{
    x = r.y;
}

void   main()
{
    X     x(5);
    Y     y(10);

    cout << "X = " << x.GetX() << '\t' << "Y = " << y.GetY() << endl;
    x.SetX(y);                               // A
    cout << "X = " << x.GetX() << '\t' << "Y = " << y.GetY() << endl;
}
```

该程序的输出为：

```
X = 5 Y = 10
X = 10      Y = 10
```

在例 11.2 中，由于将类 X 的成员函数 SetX（）说明为类 Y 的友元，这就使得类 X 可以很方便地访问类 Y 中的私有成员 y。若没有友元机制，就只能在类 X 中将成员函数 SetX（）定义为

```
void  SetX(int a)  {x = a;}
```

且例中的行 A 就得改写为

```
x.SetX(y.GetY());
```

这实际上进行了两次函数调用，而行 A 则只需一次函数调用。

若将一个类 X 中所有的成员函数都说明成另一个类 Y 的友元，则可以不必在类 Y 中将类 X 的所有成员函数一一说明，而只需简写为：

```
class Y    {
      // 类 Y 的成员说明
      friend class X;
};
```

这实际上就是将整个类 X 说明为类 Y 的友元。当一个类被说明为另一个类的友元时，前者所有的成员函数都可以访问后者中的任一成员。

例 11.3 将类作为友元。

```
//      EX11_3.CPP

#include      <iostream.h>

class  X      {
       private:
              int      x;
       public:
              X(int a = 0)    {x = a;}
              int      GetX() {return x;}
              friend class    Y;                    // A
};

class  Y      {
       private:
              int      y;
       public:
              Y(int b = 0)    {y = b;}
              void     Out(X &r)
              {
                     cout << "X = " << r.x << '\t' << "Y = " << y << endl;
              }
};

void   main()
{
       X      x(5);
       Y      y(10);

       y.Out(x);
}
```

该程序的输出为

```
X = 5 Y = 10
```

在例 11.3 里，将类 Y 说明为类 X 的友元，从而使得类 Y 中所有的成员函数有权访问类 X 中的私有成员。若没有这样的说明，则在 Y :: Out（）函数中对类 X 的私有成员的访问就

是非法的，只能将其输出语句改为

```
cout << "X = " << r.GetX() << '\t' << "Y = " << y << endl;
```

这就增加了一次函数调用。

由于友元破坏了类的封装性，所以，只有在两个类具有极密切的关系时，才将一个类说明为另一个类的友元。另外，这种说明不具有交换性。也就是说，虽然类 Y 是类 X 的友元，但类 X 却并不是类 Y 的友元，在类 X 中不能访问类 Y 中的私有成员。

在例 11.3 中，虽然在类 X 的说明中用到了类 Y（见程序中行 A），但在这里属于引用性说明，因此不需要超前说明。

应说明的是：在类体系中，友元关系是不继承的。例如，在例 11.3 中增加一个类说明：

```
class Z : public Y {
        int   z;
    //...
    public:
        //...
        int   Sum(X &r) {return r.x + y + z;}
        void  Out(X &r)
        {
            Y :: Out(r);
            cout << "Z = " << z <<endl;
        }
};
```

这里，成员函数 Z :: Sum（）的实现部分就是错误的，因为它直接访问了类 X 的私有成员 x。虽然类 Y 是类 X 的友元，但它的派生类 Z 并不是类 X 的友元。类 Z 的对象无权访问类 X 的私有成员，而成员函数 Z :: Out（）则是正确的，它是借助其基类 Y 来访问类 X 中的私有成员。由于该函数是函数 Y :: Out（）的重载，所以函数体中对其基类的同名函数的访问必须使用作用域限定，否则将是一个无限递归的函数。

11.2 虚函数

多态性是面向对象程序设计技术的关键概念之一。多态性概念是用来描术过程的，利用多态性可以用同一个函数名访问一个函数的不同实现。

C++语言支持编译时多态性和运行时多态性。所谓编译时多态性是指在编译器对源程序进行编译时就可以确定所调用的是哪一个函数，编译时多态性通过重载（函数重载和运算符重载）来实现；而运行时多态性则是指在程序运行过程中根据具体情况来确定调用的是哪一个函数，运行时多态性通过继承和虚函数来实现。

11.2.1 虚函数

一个虚函数就是一个在某基类中说明为 virtual 并在一个或多个派生类中被重新定义的成员函数。说明一个虚函数的一般形式为

virtual type ***func_name(<arg_List>);***

一个函数一旦被说明为虚函数，则无论说明它的类被继承了多少层，在每一层派生类中该函数都保持 virtual 特性。因此，在派生类中重新定义该函数时，不再需要关键字 virtual。但习惯上，为了提高程序的可读性，常常在每层派生类中都重复地使用 virtual 关键字。

例 11.4 虚函数。

```
//      EX11_4.H

#include      <iostream.h>

class  B      {
       public:
              virtual void    virfunc()       {cout << "Class B\n";}
};

class  D : public B   {
       public:
              virtual void    virfunc()       {cout << "Class D\n";}
};

class  E : public B   {
       public:
              virtual void    virfunc()       {cout << "Class E\n";}
};


//      EX11_4.CPP

#include      "ex11_4.h"

void   main()
{
       B      b, *pb;
       D      d;
       E      e;

       pb = &b;
       pb->virfunc();
       pb = &d;
       pb->virfunc();
       pb = &e;
       pb->virfunc();
}
```

该程序的输出为：

```
Class B
Class D
Class E
```

在本例中，使用了一个指向基类 B 的指针 pb 来调用虚函数 virfunc（）。当 pb 取类体系中不同类对象的地址时，调用的就是不同类中的虚函数，这就是运行时的多态性。

使用虚函数时应注意：

1. 在类体系中访问一个虚函数时，应使用指向基类类型的指针或对基类类型的引用，以满足运行时多态性的要求。当然也可以像调用普通成员函数那样利用对象名来调用一个虚函数。

例 11.5 利用对象名调用虚函数。

```
//      EX11_5.CPP

#include      "ex11_4.h"

void   main()
{
       B      b;
       D      d;
       E      e;

       b.virfunc();
       d.virfunc();
       e.virfunc();
}
```

该程序的输出与例 11.4 的输出完全相同。然而，本例属于编译时多态性，因为该程序在编译时已能确定其 3 个函数调用分别调用的是哪一个 virfunc（）。只有利用指向基类类型的指针或对基类类型的引用调用一个虚函数时，才能实现运行时多态性。这是因为指针所指的或引用所代表的内存地址只有在程序运行过程中才能确定。

2. 在派生类中重新定义虚函数时，必须保证该函数的值和参数与基类中的说明完全一致，否则就属于重载（参数不同）或是一个错误（返回值不同）。

3. 若在派生类中没有重新定义虚函数，则该类的对象将使用其基类中的虚函数代码。例如，若将 EX11_4.H 中类 E 的定义改为

```
class  E : public B {};
```

则例 11.5 的输出将变为：

```
Class B
Class D
Class B
```

4. 虚函数必须是类的一个成员函数，不能是友元，但它可以是另一个类的友元。另外，虚函数不得是一个静态成员（将在 11.3 节中介绍）。

5. 析构函数可以是 virtual 的，但构造函数则不得是虚函数。一般地讲，若某类中定义有虚函数，则其析构函数也应当说明为虚函数。特别是在析构函数需要完成一些有意义的操作——比如释放内存——时，尤其应当如此。

11.2.2 纯虚函数

在编写面向对象的程序时，并非必须使用虚函数。然而，利用虚函数可以使所设计的软件系统变得更加灵活，提高了代码的重用性。同时，虚函数为一个类体系中所有子类的同一行为提供了统一的接口，这就使得程序员在使用一个类体系时只须记住一个接口而不是多个接口。另外，接口与实现的分离支持了类库的建立。若能正确地实现这些库，则它们将操作一个公共的接口，可以用来派生自己的类以满足特定的需要。正因为如此，有时在说明一个基类时无法为虚函数定义其具体实现，这时可以将它说明为纯虚函数，其具体实现留给派生类来定义。说明一个纯虚函数的一般形式为

virtual type　　func_name(<arg_list>) = 0;

纯虚函数是构成抽象类的一个因素，包含有纯虚函数的类为抽象类。

例 11.6 包含有纯虚函数的类。

```
//      EX11_6.H

#if !defined   _EX11_6_H_
#define        _EX11_6_H_

class   Object {
        public:
                Object()        {}
                virtual ~Object()     {}                    // 虚析构函数
                virtual int     IsEqual(Object&) = 0;
                virtual void    Show() = 0;
};

#endif
```

在头文件 EX11_6.H 里，说明了一个类 Object。该类中包含有两个纯虚函数 IsEqual（）和 Show（），前者用来判断两个对象是否相等而后者用来显示对象的内容。然而，从类 Object 的定义可以看出，它根本就没有数据成员，因而也就无法对该类的对象进行比较、无内容可供显示。这就是将它们说明成纯虚函数的原因。因为 Object 是一个抽象类，所以在程序中不能说明该类的对象。那么，说明这样的一个不含任何数据成员的类到底有什么用处呢？说明这样一个类是为了将它作为我们所设计的软件系统中大多数类的基类，并尽可能地为由它所派生出来的类体系提供统一的接口。

有了这样的公共基类，就可以设计出一些通用的操作类，比如对链表的操作。

例 11.7 定义一个双链表类。

```
//      EX11_7.H

#if !defined   _EX11_7_H_
#define        _EX11_7_H_

#include       "ex11_6.h"
```

```
class   Node  {                                          // 结点类
        private:
                Object *Info;                          // 结点的数据域
                Node   *Prev, *Next;                   // 结点的链域
        public:
                Node() : Info(0), Prev(0), Next(0)   {}
                Node(Node &node)
                {
                        Info = node.Info;
                        Prev = node.Prev;
                        Next = node.Next;
                }
                void    FillInfo(Object *obj) {Info= obj;}   // 填充数据域
                friend class    List;                  // 指定类 List 为该类的友元
};

class   List    {                                      // 双链表类
        private:
                Node   *Head, *Tail;                   // 链表的头指针和尾指针
        public:
                List() : Head(0), Tail(0)       {}
                ~List() {DestroyList();}
                void    AppendNode(Node*);
                Node   *DeleteNode(Node*);
                Node   *LookUp(Object&);
                void    ShowList();
                void    DestroyList();
};

#endif


//      EX11_7.CPP

#include      "ex11_6.h"
#include      "ex11_7.h"

void    List :: AppendNode(Node *node)
// 将指针 node 所指的结点添加到链表的尾端
{
        if(Head == 0) {                        // 若 Head 为空指针，则说明该链表为一空表
                Head = Tail = node;            // 因此，表头指针与表尾指针均指向该结点

                return;
        }
        Tail->Next = node;                     // 若链表不空，则将该结点链在链表尾部
        node->Prev = Tail;                     // 新结点的前趋为原链表中的尾结点
        Tail = node;                           // 修改尾指针
}
```

```
Node   *List :: DeleteNode(Node *node)
// 从链表中删除由 node 所指的结点并返回该结点
{
        node->Prev->Next = node->Next;          // 修改欲删结点前趋的后指针
        node->Next->Prev = node->Prev;          // 修改欲删结点后继的前指针
        node->Prev = node->Next = 0;

        return node;                            // 返回被欲删除的结点
}

Node   *List :: LookUp(Object &obj)
// 在链表中查找内容与 obj 相同的结点。找到时返回指向该结点的指针；否则返回一个空指针
{
        Node   *pn = Head;

        while(pn)      {                        // 从表头起，逐一查找各结点
                if(pn->Info->IsEqual(obj))      // 若 pn 所指结点与欲找对象的内容相等
                        return pn;              // 则返回指向该结点的指针
                pn = pn->Next;                  // 否则，pn 指向下一结点
        }

        return 0;                               // 未找到，返回空指针
}

void    List :: ShowList()
// 显示整个链表中各结点的内容
{
        Node   *pn = Head;

        while(pn)      {                        // 从表头起逐一显示各结点的内容
                pn->Info->Show();               // 显示当前结点的内容
                pn = pn->Next;                  // 下一个结点
        }
}

void    List :: DestroyList()
// 销毁整个链表
{
        Node   *pn, *qn;

        pn = Head;
        while(pn)      {
                delete pn->Info;                // 删除结点的数据域
                qn = pn;                        // 保存指向当前结点的指针
                pn = pn->Next;                  // pn 指向下一个结点
                delete qn;                      // 删除当前结点
        }
}
```

例 11.7 给出的是一个简化了的双链表类，一个完整的双链表类还应包括前插入（在表中任意指定结点的前面插入一个结点）、后插入（在表中任意指定结点的后面插入一个结点）、前删除、后删除、遍历（逐一访问表中所有结点），等等。关于双链表的操作已超出了本书的内容，有兴趣的读者可参阅有关“数据结构”的资料。

上述的链表可以用来存放类 Object 的任何派生类的对象，下面给出一个演示使用双链表类的例子。

例 11.8 使用双链表。

```
//      EX11_8.CPP

#include      <iostream.h>
#include      "ex11_6.h"
#include      "ex11_7.h"

class   A_Obj : public Object       {
        private:
                int   Num;
        public:
                A_Obj(int n = 0) : Num(n)   {}
                void   SetNum(int a) {Num = a;}
                virtual int     IsEqual(Object&);
                virtual void    Show()
                {
                        cout << "Num = " << Num << endl;
                }
};

int     A_Obj :: IsEqual(Object &obj)
{
        A_Obj &a = (A_Obj&)obj;

        return (Num == a.Num);
}

void    main()
{
        A_Obj *po, key;
        Node  *pn, node;
        List    list;

        for(int i = 1; i < 6; i ++)      {          // 建立一个包含有 5 个结点的链表
                po = new A_Obj(i);                  // 创建一个 A_Obj 类型的对象
                pn = new Node;                      // 创建一个新结点
                pn->FillInfo(po);                   // 填充结点的数据域
                list.AppendNode(pn);                // 将结点添加到链表中
        }
        list.ShowList();                            // 显示链表
        key.SetNum(3);                              // 设定查找关键字为 3
```

```
    pn = list.LookUp(key);                    // 在链表中查找与关键字相匹配的结点
    if(pn)                                    // 若找到，则从链表中删除该结点
        pn = list.DeleteNode(pn);
    cout << endl;
    list.ShowList();                          // 显示删除过的链表
    list.AppendNode(pn);                      // 将刚才删除的结点加入到链表尾部
    cout << endl;
    list.ShowList();                          // 显示修改过了的链表
}
```

该程序的输出为：

```
Num = 1
Num = 2
Num = 3
Num = 4
Num = 5

Num = 1
Num = 2
Num = 4
Num = 5

Num = 1
Num = 2
Num = 4
Num = 5
Num = 3
```

以上各模块的功能及执行过程在源程序中已注释清楚，请读者对照例 11.8 的输出自行分析。

上例中的类 A_Obj 由类 Object 派生而来，它根据自身的数据结构定义了从其基类继承来的虚函数 IsEqual（）和 Show（）的具体实现，使得这两个函数可以正确地操作类 A_Obj 的对象。类 Object 的不同的派生类均可以（通常也是必须）定义自己的由基类继承来的虚函数，以使它们适应本类的数据结构，而在应用程序中则可以利用相同的接口实现不同的操作。

例 11.9 使用双链表的又一个例子。

```
//      EX11_9.CPP

#include      <iostream.h>
#include      <string.h>
#include      <stdio.h>
#include      "ex11_6.h"
#include      "ex11_7.h"

class   B_Obj : public Object       {
        private:
```

```
        char *Str;
    public:
        B_Obj() : Str(0)        {}
        B_Obj(char*);
        ~B_Obj();
        void    SetStr(char*);
        int     IsEqual(Object&);
        void    Show()          {cout << Str << endl;}
};

B_Obj :: B_Obj(char *s)
{
    Str = new char[strlen(s) + 1];
    strcpy(Str, s);
}

B_Obj :: ~B_Obj()
{
    delete Str;
}

void    B_Obj :: SetStr(char *s)
{
    if(Str != 0)
        delete Str;
    Str = new char[strlen(s) + 1];
    strcpy(Str, s);
}

int     B_Obj :: IsEqual(Object &obj)
{
    B_Obj &b = (B_Obj&)obj;

    return (strcmp(Str, b.Str) == 0);
}

void    main()
{
    B_Obj *po, key;
    Node  *pn, node;
    List   list;
    char   str[81];

    do     {
        gets(str);                          // 输入一个字符串
        if(*str == '\0')                    // 若输入为一空串，则退出循环
            break;
        po = new B_Obj(str);
        pn = new Node;
        pn->FillInfo(po);
```

```
        list.AppendNode(pn);
    } while(1);

    list.ShowList();
    cout << "Enter a key string: ";
    gets(str);                                  // 输入欲查找的字符串
    key.SetStr(str);
    pn = list.LookUp(key);
    if(pn)
        pn = list.DeleteNode(pn);
    cout << endl;
    list.ShowList();

    list.AppendNode(pn);
    cout << endl;
    list.ShowList();
}
```

本例中的类 B_Obj 也是类 Object 的派生类，由于它的数据结构与类 A_Obj 不同，因此虚函数 IsEqual（）和 Show（）的具体实现也与类 A_Obj 不同。然而，比较例 11.8 与例 11.9 可以看出，无论类 Object 的派生类的数据结构怎样变化，类 Node 和 List 均无需作任何改动。这又一次体现了 OOP 技术所支持的代码重用机制。

实际上，本书以前所给出的任何一个类，只需将它们改写为类 Object 的派生类，并定义相应虚函数的具体实现，都可以将它们存入链表 List 中。作为练习，请读者将类 Person 用于链表中。

11.3 静态成员

类相当一个数据类型，每当说明一个某类的对象时，系统就为该对象分配一块内存用来存放类中的所有成员。也就是说，在每一个对象中都存放有其所属类中所有成员的拷贝。但在有些应用中，希望程序中同类对象共享某个数据。为此，一个解决办法就是将所要共享的数据说明为全局变量，但这将破坏数据的封装性；较好的解决办法是将所要共享的数据说明成类的静态成员。

11.3.1 静态数据成员

类中用关键字 static 修饰的数据成员叫做静态数据成员。说明一个静态数据成员的方法与说明一个普通静态变量一样，只不过前者是在一个类中说明的。例如：

```
class C     {
            int     d;
            static int     s;
    //...;
}   c1, c2;

int     C :: s;         // A
```

这里，类 C 中包含有两个数据成员 d 和 s。其中 d 为普通数据成员，在对象 c1 和 c2 中都存在有各自的该数据成员的副本；而 s 则为静态数据成员，所有类 C 对象中的该成员实际上是同一个变量。C++编译器将静态数据成员存放在静态存储区，该存储区中的数据为类的所有对象所共享。

在类中说明的静态数据成员属于引用性说明，必须在文件作用域中的某个地方对静态数据成员进行定义性说明，且仅能说明一次。上例中的行 A 就是对静态数据成员 s 的定义性说明。在对静态数据成员进行定义性说明时，也可以对其进行初始化。比如，可将上面的行 A 改写为：

```
int    C :: s = 5;
```

这样，就在定义 C::s 的同时将其值初始化为 5。

例 11.10 使用静态数据成员。

```
//      EX11_10.CPP

#include      <iostream.h>

class  C      {
              int     d;
              static int       s;
       public:
              C(int a = 0)
              {
                     d = a;
                     s ++;
              }
              int     GetD() {return d;}
              int     GetS() {return s;}
              void    SetD(int a)    {d = a;}
};

int    C :: s = 0;

void   main()
{
       C      c[5];
       int    i;

       for(i = 0; i < 5; i ++)
              c[i].SetD(i + 1);
       for(i = 0; i < 5; i ++)
              cout << "c[" << i << "].d = " << c[i].GetD() << '\t'
                     << "c[" << i << "].s = " << c[i].GetS() << endl;
       cout << "There are " << c[0].GetS() << " objects created.\n";
}
```

该程序的输出为：

```
c[0].d = 1    c[0].s = 5
```

```
c[0].d = 2    c[0].s = 5
c[0].d = 3    c[0].s = 5
c[0].d = 4    c[0].s = 5
c[0].d = 5    c[0].s = 5
There are 5 objects created.
```

本例中，类 C 中的静态成员 s 实际上是用来统计在程序中创建了多少个类 C 的对象的。该成员的初值被初始化为 0，每当创建一个类 C 的对象，构造函数 C :: C（）就将其值加 1。在本例中，没有定义为 C :: s 置数的成员函数，这里也没有这个必要。若不将该数据成员说明为静态的，则由于每个对象中将分别保存着一个其值各不相同的 s 成员，则不仅造成了存储单元的浪费，而且为了使它们的值统一，将导致操作变得复杂。

11.3.2 静态成员函数

与数据成员一样，成员函数也可以被说明成静态的。比如：

```
class C     {
            //...;
      public:
            static int     sfunc();
};

int     C :: sfunc()
{
      // func_body;
}
```

这里，成员函数 C :: sfunc（）就是类 C 中的一个静态成员函数。注意：在上述对该函数定义时，没有加关键字 static。这是由于关键字 static 不是数据类型的组成部分，因此，在类说明之外定义静态成员函数时，不得使用该关键字。静态成员函数与其他成员函数一样，也可以说明成内联的，其说明方式也与普通成员函数一样。但静态成员函数不得说明为虚函数。

由于静态成员函数没有 this 指针，因此，静态成员函数只直接访问类中的静态成员，若要访问类中的非静态成员时，必须借助对象名或指向对象的指针。

例 11.11 使用静态成员函数。

```
//      EX11_11.CPP

#include      <iostream.h>

class  C      {
              int     d;
              static int       s;
       public:
              C(int a = 0)
              {
```

```
            d = a;
            s ++;
        }
        static int      GetD(C &r)   {return r.d;}
        static int      GetS() {return s;}
        void   SetD(int a)    {d = a;}
};

int  C :: s = 0;

void  main()
{
    C     c[5];                    // A
    int   i;

    for(i = 0; i < 5; i ++)
        c[i].SetD(i + 1);
    for(i = 0; i < 5; i ++)
        cout << "c[" << i << "].d = " << C :: GetD(c[i]) << '\t'
             << "c[" << i << "].s = " << C :: GetS() << endl;
    cout << "There are " << C :: GetS() << " objects created.\n";
}
```

该程序的输出与例 11.10 完全相同。

注意：例中对静态成员函数的访问采用了成员名限定，实际上，完全可以使用对象名访问静态成员函数。比如，可将例中的：

```
C :: GetD(c[i])
```

改写为

```
c[i].GetD(c[i])
```

其作用完全相同。但由于静态成员为类的所有对象所共同拥有，它们并不属于某一个对象，所以采用成员名限定比使用对象名来访问静态成员在逻辑上更为合理。

由于类中所有非静态成员函数均可以直接访问静态的和非静态的数据成员，并且在 C++ 语言的具体实现中，为了节省内存而将一个类所有对象的成员函数只保存一个拷贝，因此，静态成员函数一般讲仅有其逻辑上的意义——一个属于所有对象所共有的成员函数。通常，若类中说明有静态数据成员，则将访问该成员的函数说明成静态的。

静态成员具有外部连接属性，因此，即使还没有建立对象也可以访问类中的公有静态成员。例如，可以在例 11.11 行 A 前边加一条语句：

```
cout << "Before objects creating the s is " << C :: GetS() << endl;
```

用以显示静态数据成员 s 的初始值。

11.4 const，volatile 对象和 const，volatile 成员函数

与普通变量一样，可以使用关键字 const 和 volatile 来修饰一个对象。然而，与普通变量

不同的是：通过这样的对象只能访问其所在类中用同样关键字所修饰的成员函数。也就是说，用 const 修饰的对象（称 const 对象）只能访问类中用 const 修饰的成员函数（称 const 成员函数）；用 volatile 修饰的对象（称 volatile 对象）只能访问类中用 volatile 修饰的成员函数（称 volatile 成员函数）。

说明一个 const 或 volatile 对象的一般形式为

key_word class_name object_name;

其中：key_word 即关键字 const 或 volatile；class_name 为一个已存在的类名；而 object_name 则为程序中所说明的对象名。例如：

```
const Person      per;
```

就说明了一个类型为类 Person 的 const 对象 per。

而说明一个 const 或 volatile 成员函数的一般形式为

type func_name(<arg_list>) key_word;

例如：

```
int   Func(int, char*)   const;
```

就说明了一个名为 Func 的 const 成员函数。注意：上述说明必须在一个类说明中进行，否则就不是成员函数。另外请注意：说明 const 或 volatile 成员函数时关键字的位置是在函数说明的尾部，若将关键字放在函数说明的首部，则是对函数值的修饰。比如：

```
const int    Func(int, char*);
```

就说明的是一个返回常整型量的函数 Func（），而不是一个 const 函数。

例 11.12　const、volatile 对象和成员函数。

```
//      EX11_12.CPP

#include      <iostream.h>

class  C      {
              int     x;
       public:
              C(int a = 0) : x(a)     {}
              int     NormFunc()  {return ++ x;}
              int     ConFunc()   const   {return x + 1;}
              int     VolFunc()   volatile        {return ++ x;}
};

void   main()
{
       C      c(1);
       const C        ConC(3);
       volatile C     VolC(5);

       cout << c.NormFunc() << ' ';
```

```
    cout << c.ConFunc() << ' ';
    cout << c.VolFunc() << endl;
    cout << ConC.ConFunc() << endl;
    cout << VolC.VolFunc() << endl;
}
```

该程序的输出为：

```
2 3 3
4
6
```

本例在类 C 中定义了一个一般成员函数 NormFunc（），一个 const 成员函数 ConFunc（）和一个 volatile 成员函数 VolFunc（）。程序中还说明了 3 个类 C 的对象：普通对象 c，const 对象 ConC 和 volatile 对象 VolC。从例中可以看出，通过 c 可以访问类中的任何一个成员函数，通过 ConC 只能访问成员函数 ConFunc（），通过 VolC 只能访问成员函数 VolFunc（）。以下的函数调用都将导致一个编译错误：

```
ConC.NormFunc();
ConC.VolFunc();
VolC.NormFunc();
VolC.ConFunc();
```

应当说明的是，有的编译器为了使用户在编程时不致于对已存在的代码做较大的改动，对上述的函数调用仅给出警告信息，仍能生成正确的可执行文件。

由于关键字 const 和 volatile 属于数据类型的组成部分，因此，若在类外定义 const 成员函数和 volatile 成员函数时，则相应的关键字必须存在。比如：

```
int    C :: ConFunc()    const
{
    return x + 1;
}
```

若类中还说明有与 const 成员函数或 volatile 成员函数同名的普通成员函数时，即使其参数表完全相同，也属于重载。在这种情况下，该类的普通对象所调用的是该重载的成员函数，而不再是同名的 const 或 volatile 成员函数。例如，将例 11.12 中的类 C 改写为：

```
class C    {
            int    x;
    public:
            C(int a = 0) : x(a);
            int    NormFunc(){return ++ x;}
            int    ConFunc()  {return x;}
            int    VolFunc()  {return x;}
            int    ConFunc()  const {return x + 1;}
            int    VolFunc()  volatile      {return ++ x;}
};
```

则例 11.12 的输出将变为：

```
2 2 2
4
6
```

也可以同时用这两个关键字来修饰同一个对象或成员函数，说明时两个关键字的先后次序无规定。这种用两个关键字修饰的对象只能调用用两个关键字所修饰的成员函数；而用两个关键字所修饰的成员函数则还可以被 const 对象或 volatile 对象所调用。

const 对象和 volatile 对象的作用与 const 变量的 volatile 变量的作用一样，这里不再赘述。

如同关键字 const 的意义一样，一个 const 成员函数无权修改类中的任何一个数据成员的值。比如，将成员函数 ConFunc（）的函数定义改为

```
int    C :: ConFunc()    const {return ++ x;}
```

就属于非法操作。

为了便于使用，为一个类说明一些 const 和 volatile 成员函数常常是很有必要的。比如，为了改善传递参数时的时间和空间开销而用对类类型的引用作为函数的参数。但引用参数属于引用调用，函数中对形参的修改将会影响实参的值。为了防止函数对形参的修改，可将参数说明为 const 的，以便在编译时能发现非法的修改操作。

例 11.13 一个简单的字符串类。

```
//      EX11_13.H

#if !defined   _EX11_13_H_
#define        _EX11_13_H_

#include       <iostream.h>

class   String {
        private:
                int     Length;
                char    *Str;
                void    CopyStr(char*, int);
        public:
                String() : Length(0), Str(0)   {}
                String(const String&);
                String(char*);
                char    *isIn(char)    const;
                char    *GetStr(char*);
                int     GetLen()       {return Length;}
                void    Show()         {cout << Str << endl;}
                ~String() {delete Str;}
};

#endif


//      EX11_13A.CPP
```

```
#include      <string.h>
#include      "ex11_13.h"

void    String :: CopyStr(char *str, int len)
{
        Length = len;
        Str = new char[Length + 1];
        strcpy(Str, str);
}

String :: String(const String &str)
{
        CopyStr(str.Str, str.Length);
}

String :: String(char *str)
{
        CopyStr(str, strlen(str));
}

char    *String :: isIn(char ch)        const
// 检查对象中是否存在 ch 所代表的字符，若是，则返回对象中第一个匹配字符的地址；否则返回空指针
{
        char    *cp = Str;

        while(*cp)    {
                if(*cp == ch)
                        return cp;
                cp ++;
        }

        return 0;
}

char    *String :: GetStr(char *buff)
{
        strcpy(buff, Str);

        return buff;
}


//      EX11_13.CPP

#include      <iostream.h>
#include      "ex11_13.h"

void    main()
{
        int     isContent(const String&, char);
```

```
    String  str("This is a string.");

    str.Show();
    if(str.isIn('i') != 0)
        cout << "The 'i' is in string.\n";
    else
        cout << "The 'i' is not in string.\n";

    if(isContent(str, 'j'))
        cout << "The 'i' is in string.\n";
    else
        cout << "The 'j' is not in string.\n";
}

int    isContent(const String &s, char c)
{
    if(s.isIn(c))          // A
        return 1;
    else
        return 0;
}
```

该程序的输出为：

```
This is a string.
The 'i' is in string.
The 'j' is not in string.
```

本例在类 String 中说明了一个 const 成员函数 isIn（），若将该函数说明成一个一般的成员函数，则在函数 isContent（）中行 A 对该函数的调用将导致一个错误。其原因请读者自行分析。

11.5 指向类成员的指针

C++语言提供了一种称为指向类成员的指针（简称成员指针）的特殊指针，该指针用来指向指定类中的成员。

说明一个成员指针的一般形式为

type class_name :: *pointer_name;

与说明一个普通指针相比，说明成员指针多了一个成员名限定 class_name ::。比如

```
char  Person :: *chPtr;
```

就说明了一个指向类 Person 中成员的字符型成员指针。注意：成员指针不是类中的一个成员，因此成员指针应在类外说明。

若类中某数据成员 member 的数据类型与成员指针的类型相同，则可以利用以下的方式将该数据成员的地址赋给成员指针：

pointer_name = &class_name :: member;

比如：

chPrt = &Person :: Name;

就将类 Person 的成员 Name 的首地址赋给了成员指针 chPtr。

应当说明的是：通过这样的赋值，chPtr 取得的并不是 Name 的绝对地址，而是成员 Name 在所有类 Person 的对象中的偏移量，即 Name 距类 Person 所有对象首地址的字节数。顺便说一句，尽管一个类的各对象在内存中的首地址各不相同，但其每一个成员在对象中的偏移量总是一个常量。

访问一个成员指针所指的内容时，需要用到成员访问运算符“.*”和“->*”。通过成员指针访问它所指的内容时，必须指定一个对象，以明确是对哪一个对象中的数据成员的访问。若所指定的对象是以对象名或引用所标识的，则应使用运算符“.*”；若所指定的对象是以指向对象的指针所标识的，则应使用运算符“->*”。

例 11.14 使用成员指针。

```
//      EX11_14.H

class  C      {
       public:
              int     x;
              C(int a = 0) : x(a)     {}
              void    SetX(int a)     {x = a;}
              int     GetX(){return x;}
};


//      EX11_14.CPP

#include      <iostream.h>
#include      "ex11_14.h"

void   main()
{
       C      c(5), *cp = &c;
       int    C :: *iPtr;

       iPtr = &C :: x;                    // A
       cout << c.*iPtr << endl;           // B
       cout << cp->*iPtr << endl;         // C
}
```

该程序的输出为：

```
5
5
```

请读者自行分析程序中行 B 与行 C 中使用不同运算符的原因。

应当指出的是：成员指针只能用来指向类中的公有成员。若将例中的数据成员 x 说明为 private 的，则程序中的行 A 将导致一个编译错误。

也可以说明指向类成员函数的函数指针（简称成员函数指针）。说明一个成员函数指针的一般形式为

type (class_name :: *pointer)(<arg_list>);

为成员函数指针赋值具有以下的一般形式：

pointer = class_name :: func_name;

其中：func_name 为类 class_name 中的一个返回值和参数表与成员函数指针 pointer 完全相同的成员函数。

例 11.15 使用成员函数指针。

```
//      EX11_15.CPP

#include      <iostream.h>
#include      "ex11_14.h"

void   main()
{
       C      c(5), *cp = &c;
       int(C :: *fPtr)();
       int   i;

       fPtr = C :: GetX;
       i = (c.*fPtr)();
       cout << i << endl;
       cout << (c.*fPtr)() << endl;
       cout << (cp->*fPtr)() << endl;
}
```

该程序的输出为：

```
5
5
5
```

注意：通过成员函数指针调用一个成员函数时，括起函数指针访问的圆括号是必不可少的。另外，成员函数指针也只能用来指向类中的公有成员函数。

当一个成员函数指针指向一个虚函数，且通过指向对象的指针或对对象的引用来访问该成员函数指针时，仍发生多态性。

例 11.16 成员函数指针指向虚函数。

```
//      EX11_16.CPP

#include      <iostream.h>
#include      "ex11_4.h"
```

```
void    main()
{
        void    (B :: *fPtr)();
        B       b;
        D       d;
        E       e;

        fPtr = B :: virfunc;
        (b.*fPtr)();
        (d.*fPtr)();
        (e.*fPtr)();
}
```

该程序的输出为：

```
Class B
Class D
Class E
```

若需要使用指向类中静态成员的指针，只须使普通指针即可。

例 11.17 使用指针指向静态成员。

```
//      EX11_17

#include      <iostream.h>

class   C       {
        public:
                static int      x;

                static int      GetX() {return x;}
};

int     C :: x = 5;

void    main()
{
        int     *sp, (*sfp)();

        sp = &C :: x;
        sfp = C :: GetX;
        cout << *sp << endl;
        cout << sfp() << endl;
}
```

该程序的输出为：

```
5
5
```

由于静态成员不属于任何对象，所以通过指针访问类中的静态成员时不需要为指针指定对象。然而，由于静态成员是在类中说明的，所以将静态成员的地址赋给指针时须用成员名

限定。

习题

11-1 仿照例 11.7，将习题 9-3 所要求的通讯录用一个双链表存放。

11-2 将例 11.6 中定义的类 Object 尽可能地作进一步地完善，然后将它用作类 Person 的基类，并在 Person 及其派生类中分别完成类 Object 中虚函数的定义。

11-3 在学生类、职大学生类、教师类和研究生类中增加一个表态成员——学校名称，并定义相应的成员函数。

第 12 章　运算符重载

在非面向对象的程序设计语言中，所有运算符在语言中都已预先定义了它们的用法与意义，并且这些定义是不能改变的。若语言提供的运算符所执行的操作不能满足用户的要求，则程序员只能通过定义函数来完成特定的操作。

C++语言允许程序员重新定义已有的运算符，使其能按用户的要求完成一些特定的操作，这就是所谓的运算符重载。与函数的重载和虚函数一样，运算符重载也从一个方面体现了 OOP 技术的多态性。

12.1　运算符重载

事实上，各种算法语言中的大多数运算符已经具有运算符重载的性质。比如，加法运算符“+”，它既可以完成对整型数据的加法运算，也可以完成对实型数据的加法运算。从用户角度来看，无论是对整型数据还是对实型数据，运算符“+”的用法是完全相同的。

但是，对于面向对象的程序设计来讲，这还远远不够，它所要求的运算符重载能完成更加复杂的操作。比如，对类的对象进行算术运算，且在使用上就如同对整数进行算术运算一样方便和自然。

12.1.1　重载运算符

重载一个运算符，必须定义该运算符的具体操作。为了使程序员能像定义函数的具体操作一样来重载一个运算符，C++提供了 operator 函数。该函数具有以下的一般形式：

```
type  class_name :: operator @(<arg_list>)
{
    func_body;
}
```

其中：type 为函数的返回值，也就是运算符的运算结果值；class_name 为该运算符重载所属类的类名；而 @ 即为所重载的运算符，它可以是 C++语言中除以下 4 个运算符外的所有运算符：

::　　.　　*（递引用运算符）　　?:

这里，运算符“*”是指访问一个指针变量内容的递引用运算符，而与该运算符同形的指针说明运算符和乘法运算符则是允许重载的。

例12.1　定义一个复数类，并重载加法行算符以适应对复数运算的要求。

```
//      EX12_1.H

class   Complex      {
        private:
                double R, I;
        public:
                Complex(double r = 0, double i = 0) : R(r), I(i)      {}
                double Real()  {return R;}
                double Imag() {return I;}
                Complex      operator + (Complex&);                    // A
                Complex      operator + (double);                      // B
                Complex      operator = (Complex);
};


//      EX12_1.CPP

#include      "ex12_1.h"

Complex      Complex :: operator + (Complex &c)
{
        Complex      temp;

        temp.R = R + c.R;
        temp.I = I + c.I;

        return temp;
}

Complex      Complex :: operator + (double d)
{
        Complex      temp;

        temp.R = R + d;
        temp.I = I;

        return temp;
}

Complex      Complex :: operator = (Complex c)
{
        R = c.R;
        I = c.I;

        return *this;
}
```

在例 12.1 中，对运算符“+”进行了两次重载。其中，运算符重载 A 可以完成两个复数的加法运算，而运算符重载 B 则可以完成一个复数与一个实数的加法运算。

可以看出，运算符重载事实上就是一种函数的重载，只不过它比函数的重载多了一些限制而已。比如：它的函数名必须在 operator 后边带上一个合法的 C++运算符；它不得是类中

的静态成员函数，等等。这种重载函数的参数个数也有限制，比如，原始加法运算符要求有两个操作数，而对该运算符的重载则须有且仅能有一个参数，该参数即为加法运算符的右操作数，而左操作数则为该类的对象本身。这一点可以从函数的定义中清楚地看出。

在运算符重载函数的定义中，由程序员给出了该运算符重载的具体操作，该具体操作并不要求与所重载之运算符的意义完全相同，因为对 C++系统来讲，调用一个运算符重载只不过是调用了一个函数，与调用其他函数一样，系统并不关心该函数的具体操作是什么。比如，完全可以将运算符重载 A 定义成如下形式：

```
Complex    Complex :: operator + (Complex &c)
{
      Complex    temp;

      temp.R = R - c.R;
      temp.I = I - c.I;

      return temp;
}
```

即，将加法运算符“+”的具体操作定义成减法运算，这也不会引起任何编译错误。但这将引起逻辑上的混乱：当用户用加号对两个对象进行运算时，得到的结果却是它们的差。因此，重载一个运算符时，其具体操作应尽量与该运算符的原始含意保持一致。

12.1.2 使用运算符重载

一个重载了的运算符虽然其具体操作发生了变化，但其用法与该运算符的原始定义完全一样。

例 12.2 使用复数类。

```
//      EX12_2.CPP

#include      <iostream.h>
#include      "ex12_1.h"

void    main()
{
      Complex      c1(3, 4), c2(5, 6), c3;

      cout << "C1 = " << c1.Real() << " + " << c1.Imag() << "i\n";
      cout << "C2 = " << c2.Real() << " + " << c2.Imag() << "i\n";
      cout << "C3 = " << c3.tReal() << " + " << c3.Imag() << "i\n";
      c3 = c1;                    // 调用运算符重载 operator = (Complex&)
      cout << "C3 = " << c3.Real() << " + " << c3.Imag() << "i\n";
      c3 = c1 + c2;               // 调用运算符重载 operator + (Complex&)及 operator = (Complex)
      cout << "C3 = " << c3.Real() << " + " << c3.Imag() << "i\n";
      c3 = c3 + 3.0;              // 调用运算符重载 operator + (double)及 operator = (Complex)
      cout << "C3 = " << c3.Real() << " + " << c3.Imag() << "i\n";
}
```

该程序的输出为：

```
C1 = 3 + 4i
C2 = 5 + 6i
C3 = 0 + 0i
C3 = 3 + 4i
C3 = 8 + 10i
C3 = 11 + 10i
```

从上例可以看出：虽然运算符重载实际上是一个函数重载，但使用运算符重载就如同使用普通的运算符一样方便，并不表现为函数调用的形式。对运算符重载的具体调用由系统自动地完成，下面通过表达式 c3 = c1 + c2 来说明运算符重载的执行过程。

当编译器遇到该表达式时，就当它解释为

```
c3.operator = (c1.operator + (c2))
```

即，首先调用对象 c1 中的运算符重载：

```
Complex    Complex :: operator + (Complex&)
```

该函数的返回值为一 Complex 类型的对象；然后调对象 c3 中的运算符重载：

```
void  Complex :: operator = (Complex)
```

将上述运算结果赋给对象 c3。

这样，从用户的角度来看，就好像 C++语言本身就具有对复数的加法和赋值运算的能力一样。

应当说明的是：无论运算符重载的具体定义如何，重载了的运算符的优先级与结合性仍保持与其原始运算符的相同。

12.1.3 友元运算符

运算符重载可以是一个类的非静态成员函数，也可以是类的友元函数，后者称为友元运算符。

例 12.3 例 12.1 的另一版本。

```
//    EX12_3.H

class  Complex     {
       private:
              double R, I;
       public:
              Complex(double r = 0, double i = 0) : R(r), I(i)     {}
              double Real()  {return R;}
              double Imag() {return I;}
              friend Complex       operator + (Complex&, Complex&);
              Complex       operator = (Complex);
};
```

```
//      EX12_3.CPP

#include      "ex12_3.h"

Complex       operator + (Complex &c1, Complex &c2)
{
      Complex       temp;

      temp.R = c1.R + c2.R;
      temp.I = c1.I + c2.I;

      return temp;
}

Complex       Complex :: operator = (Complex c)
{
      R = c.R;
      I = c.I;

      return *this;
}
```

本例中，将例 12.1 中的加法运算符重载为友元运算符。由于友元没有 this 指针，所以将二元运算符重载为友元函数时必须带有两个参数，它们分别为运算符的左、右操作数。这一点可以从友元函数 operator +（）的定义中看出。另外，友元运算符中至少应该有一个参数的类型为说明该友元的类类型或是对该类的引用。将参数说明为引用类型主要是为了提高程序的运行效率。虽然使用指针也可以提高程序的效率，但是由于使用指针存在二义性，所以不得将相应的引用参数说明成指针的形式。

无论是类的成员还是友元，运算符重载的使用方法是完全一样的。

例 12.4 使用友元运算符。

```
//      EX12_4.CPP

#include      <iostream>
#include      "ex12_3.h"

{
void    main()
{
      Complex       c1(3, 4), c2(5, 6), c3;

      cout << "C1 = " << c1.Real() << " + " << c1.Imag() << "i" << endl;
      cout << "C2 = " << c2.Real() << " + " << c2.Imag() << "i" << endl;
      cout << "C3 = " << c3.Real() << " + " << c3.Imag() << "i" << endl;
      c3 = c1;
      cout << "C3 = " << c3.Real() << " + " << c3.Imag() << "i" << endl;
      c3 = c1 + c2;
      cout << "C3 = " << c3.Real() << " + " << c3.Imag() << "i" << endl;
      c3 = c3 + 3.0;
```

```
    cout << "C3 = " << c3.Real() << " + " << c3.Imag() << "i" << endl;
}
```

该程序的输出与例 12.2 完全相同。

当编译器遇到表达式 c3 = c1 + c2 时，就当它解释为

```
c3.operator = (operator + (c1, c2))
```

即，首先调用友元运算符：

```
Complex    operator + (Complex&, Complex&)
```

然后调对象 c3 中的成员函数：

```
void  Complex :: operator = (Complex)
```

将上述运算结果赋给对象 c3。

由例 12.4 可以看出，从使用角度来看，运算符重载无论是作为成员函数还是作为友元是没有什么区别的。

读者可能注意到了在文件 EX12_3.H 和 EX12_3.CPP 中没有了运算符重载：

```
operator + (double)
```

那么程序中的语句：

```
c3 = c3 + 3.0;
```

将是如何执行的呢？

当编译器遇到该语句时，它先将实型常量 3.0 通过调用类 Complex 的构造函数转换成一个 Complex 类型的对象（为此目地，类 Complex 中带有缺省参数的构造函数是非常必要的，无此构造函数将无法进行这样的转换），然后再调用相应的运算符重载函数。也就是说，编译器将该语句中赋值号右操作数中的表达式解释成：

```
operator + (c3, Complex(3.0))
```

正因为如此，若将上述语句改写为

```
c3 = 3.0 + c3;
```

则程序仍能正常运行，且运算结果不变。编译器将该语句解释为

```
c3.operator = (operator + (Complex(3.0), c3))
```

这里用一个普通数值变量来替换常量 3.0，程序仍能正运行。而在例 12.2 中，这样的语句则属于非法语句。因为在那里编译器将该语句中赋值号的右表达式解释为

```
3.0.operator + (c3)
```

但常量 3.0 并不是一个类 Complex 的对象，它不存在成员函数 operator + ()。因此，该语句将导致一个编译错误。那么编译器为什么不将常量 3.0 像友元运算符那样进行转换呢？从编译器对该表达式的解释可以看出，编译器对说明为成员函数的运算符重载解释成对某类对象中成员函数的调用，而不是对某类对象的整体访问，因此也就不存在类型转换的问题。而在

友元运算符中，对象是以参数的形式出现的，当实参与形参的类型不匹配时，编译器才对实参进行相应的类型转换。由此可见，对有些运算符来讲，说明成友元运算符使用起来将更为方便一些。

应当说明的是，以下 5 种运算符不得说明成友元运算符：

=　　()　　[]　　->　　type

其中："=" 包括所有的复合赋值运算符；而 type 则用于类型转换函数（将在 12.1.4 中介绍）。C++语言所以有此规定是由于将这些运算符重载为友元时会产生副作用。比如，设有友元赋值运算符：

```
void  operator = (Complex &c1, Complex & c2)
{
      c1.R = c2.R;
      c1.I = c2.I;
}
```

则对于类 Complex 的对象 a 和 b，赋值运算：

```
a = b;
```

会得到一个正确的结果；而对于表达式：

```
5.0 = a
```

则编译器将把它解释为

```
operator = (Complex(5.0), a)
```

这也能进行赋值。然而，由于这里的类型转换是暂时性的，赋值后常量 5.0 仍是 5.0。重要的问题在于这样的赋值本身就没有意义。

12.1.4　转换函数

有一类特殊的运算符重载，它们具有以下的一般形式：

class_name :: operator type();

其中：type 是任何一个合法的 C++类型名（包括基本数据类型和导出数据类型）以及由 const 和 volatile 修饰的表达式。

这类运算符重载不带任何参数，也无返回值类型，它们总是返回一个类型为 type 的值。这类运算符重载就叫做类型转换函数，简称转换函数。转换函数的作用是将一个类的对象转换成相应的数据类型，它们必须被定义为成员函数，不得是友元。

例12.5　转换函数。

```
//    EX12_5.H

class  Complex     {
       private:
            double R, I;
```

```
    public:
        Complex(double r = 0, double i = 0) : R(r), I(i)    {}
        double Real()  {return R;}
        double Imag() {return I;}
        friend Complex      operator + (Complex&, Complex&);
        Complex      operator = (Complex);
        operator double()    {return R;}
};


//    EX12_5.CPP

#include    <iostream.h>
#include    "ex12_5.h"

void   main()
{
    Complex     c(3, 4);

    cout << c << endl;          // A
}
```

该程序的输出为

```
3
```

由于预定义的插入运算符要求其右操作数必须具有基本数据类型或者是一个字符型指针，若在类 Complex 中没有定义转换函数，则行 A 将无法编译通过。有了以上的定义，则编译器就将行 A 解释为

```
cout << c.operator double() << endl;
```

根据该转换函数的定义，输出的就是对象 c 的实部。

由于类 Complex 中定义了转换函数，使得以下的赋值语句均成为合法的：

```
double      d;

d = c;
d = double(c);
d = (double)c;
```

应当说明的是：由于转换函数不带参数，所以在一个类中同样的转换函数不得重载一次以上。比如，在类 Complex 中再重载一个转换函数：

```
Complex :: operator double()  {return I;}
```

则会导致一个编译错误。

若一个类中重载了多个转换函数，则在使用时必须利用强制类型转换，以明确指出调用的是哪一个转换函数。

例12.6　使用多个转换函数。

```
//    EX12_6.CPP
```

```
class C     {
            int     x;
            float   y;
      public:
            C(int a = 0, float b = 0) : x(a), y(b)   {}
            operator int()  {return x;}
            operator float()        {return y;}
};

void  main()
{
      C     c(3, 4.5);

      cout << "X = " << int(c) << '\t' << "Y = " << float(c) << endl;
}
```

该程序的输出为：

```
3     4.5
```

12.1.5 赋值运算符重载

在类 Complex 中，重载了赋值运算符“=”，利用该运算符重载可以将一个 Complex 对象的数据成员赋给另一个 Complex 对象。实际上，这里完全没有必要进行这样的重载，在第 8 章中已经介绍过：同类对象之间可以整体赋值。即便没有这样的重载，利用预定义的赋值运算符仍能正确地在两个 Complex 对象之间进行赋值。

然而，利用系统预定义的赋值操作在许多场合下会出现严重的问题，例 12.7 就是一个病态的程序。

例12.7 一个病态程序。

```
//      EX12_7.CPP

#include      <iostream.h>
#include      "ex9_12.h"

void  main()
{
      Person p1("Wang Ping", 23, 'f'), p2("Li Ming", 22, 'm');
      char  str[20];

      cout << p1.GetName(str) << '\t' << p1.GetAge() << '\t' << p1.GetSex() << endl;
      cout << p2.GetName(str) << '\t' << p2.GetAge() << '\t' << p2.GetSex() << endl;
      p2 = p1;                                  // A
      cout << p1.GetName(str) << '\t' << p1.GetAge() << '\t' << p1.GetSex() << endl;
      cout << p2.GetName(str) << '\t' << p2.GetAge() << '\t' << p2.GetSex() << endl;
}
```

该程序看似没有什么问题，当行 A 执行时，预定义的赋值运算符执行了如下的操作：

```
p2.Name = p1.Name;
```

```
p2.Age = p1.Age;
p2.Sex = p1.Sex;
```

这样，就使得指针 p2.Name 和 p1.Name 都指向了同一个字符串"Wing Ping"，而指针 p2.Name 原先所指的字符串 "Li Ming" 由于其首指针的丢失，使得该串所占的动态内存单元永远得下到释放。更为严重的是，当程序结束时，系统将分别调用对象 p1 和 p2 的析构函数释放其成员 Name 所占的动态内存，由于这两个指针指的是同一块动态内存，从而导致了一个运行时间错误（Run Time Error）。

为了保证类似类 Person 这样的包含有指向动态内存之指针成员的对象能被正确地赋值，有必要为这样的类重载赋值运算符。

赋值运算符只能被重载为类的非静态成员运算符，不得重载成友元运算符。

例 12.8 为类 Person 增加几个运算符重载。

```
//      EX12_8.H

#if !defined   _EX12_8_H_
#define        _EX12_8_H_

#include       <iostream.h>

class   Person {
        private:
                char    *Name;
                unsigned        Age : 7;
                unsigned        Sex : 1;
                void    CopyName(char*);
        public:
                Person(char*, int, char);
                Person(Person&);
                Person() : Name(0), Age(0), Sex(0) {}
                ~Person()       {delete Name;}
                void    SetName(char*);
                void    SetAge(int age)         {Age = age;}
                void    SetSex(char sex)        {Sex = (sex == 'm' ? 0 : 1);}
                char    *GetName(char*);
                int     GetAge()        {return Age;}
                char    GetSex()        {return(Sex == 0 ? 'm' : 'f');}
                void    Show();
                operator        const char*()  {return Name;}       // A
                operator        int()   {return Age;}
                operator        char()  {return (Sex == 0 ? 'm' : 'f');}
                Person&         operator = (Person&);
};

#endif


//      EX12_8A.CPP

#include       <string.h>
```

```
#include        "ex12_8.h"

void    Person :: CopyName(char *str)
{
        Name = new char[strlen(str) + 1];
        strcpy(Name, str);
}

Person :: Person(char *name, int age, char sex)
{
        CopyName(name);
        Age = age;
        Sex = (sex == 'm' ? 0 : 1);
}

Person :: Person(Person &p) : Age(p.Age), Sex(p.Sex)
{
        CopyName(p.Name);
}

void Person :: SetName(char *name)
{
        if(Name)
                delete Name;
        CopyName(name);
}

char    *Person :: GetName(char *str)
{
        strcpy(str, Name);

        return str;
}

void    Person :: Show()
{
        char    str[81];

        cout << "Name: " << GetName(str) << '\t' << "Age: " << GetAge()
                << '\t' << "Sex: " << GetSex() << endl;
}

Person &Person :: operator = (Person &p)
{
        delete Name;
        CopyName(p.Name);
        Age = p.Age;
        Sex = p.Sex;

        return *this;
}
```

```
//      EX12_8.CPP

#include      <iostream.h>
#include      "ex12_8.h"

void    main()
{
        Person p1("Wang Ping", 23, 'f'), p2("Li Ming", 22, 'm');

        cout << (const char*)p1 << '\t' << int(p1) << '\t' << char(p1) << endl;
        cout << (const char*)p2 << '\t' << int(p2) << '\t' << char(p2) << endl;
        p2 = p1;                                    // A
        cout << (const char*)p1 << '\t' << int(p1) << '\t' << char(p1) << endl;
        cout << (const char*)p2 << '\t' << int(p2) << '\t' << char(p2) << endl;
}
```

在例 12.8 中，为类 Person 重载了赋值运算符。在该运算符重载的定义中，先将对象的 Name 成员所指的内容释放，然后根据参数 p 的 Name 成员所指的内容重新分配一块内存，并将 p.Name 所指的内容复制到新分配的内存中。这样就消除了上述使用预定义赋值运算符所带来的两个弊病。这里运算符重载的参数和返回值均为对类 Person 的引用，这样做是出于运行效率的考虑。另外，例中还为类 Person 重载了 3 个转换函数，使该类的对象使用起来更加灵活、方便。注意转换函数 A 的定义，由于该函数返回的是一个指针，这就为应用程序在类外修改 Name 的内容提供了机会。为了阻止这样的操作，转换函数被定义成一个指向常量的指针。

12.2 几个特殊运算符的重载

对于大多数 C++运算符，重载时一般没有什么特别的限制和要求。然而，有一些运算符由于其本身的特殊性使得对其重载也有一些特殊的要求。这些运算符主要有：**增量减量运算符**、下标运算符、函数调用运算符、成员访问运算符，以及 new 和 delete 运算符。

12.2.1 增量减量运算符

增量运算符“++”和减量运算符“--”由于存在前置与后置的区别，因而对它们重载的形式也就有所差异。下面以增量运算符为例进行介绍，其结论同样适合减量运算符。

将一个增量运算符重载成一个前置的成员运算符时，应具有以下的一般形式：

type class_name :: operator ++ ();

而将一个增量运算符重载成一个后置的成员运算符时，则应具有以下的一般形式：

type class_name :: operator ++ (int);

例 12.9 增量运算符成员函数。

```
//      EX12_9.CPP

#include      <iostream.h>
```

```
class  Demo {
            int     x;
       public:
            Demo(int i = 0) : x(i) {}
            int     GetX() {return x;}
            int     operator ++ ()          {return ++ x;}              // 前置增量
            int     operator ++ (int)       {return x ++;}              // 后置增量
};

void   main()
{
       Demo d(5);
       int     i;

       cout << "Original d.x = " << d.GetX() << endl;
       i = d ++;
       cout << "After d ++, d.x = " << d.GetX() << " and i = " << i << endl;
       i = ++ d;
       cout << "After ++ d, d.x = " << d.GetX() << " and i = " << i << endl;
}
```

该程序的输出为：

```
Original d.x = 5
After d ++, d.x = 6 and i = 5
After ++ d, d.x = 7 and i = 7
```

注意：当增量运算符被重载为后置时，函数的参数为一个无名的的整型参数，这在 C++ 语言中是合法的。实际上，该参数在这里仅仅表明在调用该函数时运算符应放在操作数的后边，参数本身在函数体中并不使用，因此也就没有必要给出它的名字。应当说明的是，如果给出了参数名，反而编译器会给出一个“参数某某从未使用”这样的警告信息。

将一个增量运算符重载为友无函数时，应具有以下的一般形式：

```
type   operator ++ (class_name&);          // 前置
type   operator ++ (class_name&, int);     // 后置
```

例 12.10 例 12.9 的另一版本。

```
//     EX12_10.CPP

#include     <iostream.h>

class  Demo {
            int     x;
       public:
            Demo(int i = 0) : x(i) {}
            int     GetX() {return x;}
            friend int      operator ++ (Demo&);
            friend int      operator ++ (Demo&, int);
};
```

```
int     operator ++ (Demo &d)
{
        return ++ d.x;
}

int     operator ++ (Demo &d, int)
{
        return d.x ++;
}

void    main()
{
        Demo  d(5);
        int     i;

        cout << "Original d.x = " << d.GetX() << endl;
        i = d ++;
        cout << "After d ++, d.x = " << d.GetX() << " and i = " << i << endl;
        i = ++ d;
        cout << "After ++ d, d.x = "  << d.GetX() << " and i = " << i << endl;
}
```

该程序的输出与例 12.9 相同。

重载了的增量运算符函数的返回值可以是任何合法的 C++数据类型，有时，为了使用上的方便，常常将运算符重载的返回值说明成类类型或者对类类型的引用。

例 12.11 三维坐标类。

```
//      EX12_11.CPP

#include      <iostream.h>

class   Three_D       {
                int     x, y, z;
        public:
                Three_D(int i = 0, int j = 0, int k = 0) : x(i), y(j), z(k)       {}
                friend Three_D        &operator ++ (Three_D&);
                friend Three_D        operator ++ (Three_D&, int);
                void    Show()
                {
                        cout << "X = " << x << '\t' << "Y = " << y << '\t' << "Z = " << z << endl;
                }
};

Three_D       &operator ++ (Three_D &t)
{
        t.x ++;
        t.y ++;
        t.z ++;

        return t;
}
```

```
Three_D      operator ++ (Three_D &t, int)
{
      Three_D      temp = t;

      t.x ++;
      t.y ++;
      t.z ++;

      return temp;
}

void   main()
{
      Three_D      t1(3, 4, 5), t2;

      cout << "Original T1: ";
      t1.Show();

      t2 = ++ t1;                          // A
      cout << "After ++ t1, T1: ";
      t1.Show();
      cout << "\tand T2: ";
      t2.Show();

      t2 = t1 ++;                          // B
      cout << "After t1 ++, T1: ";
      t1.Show();
      cout << "\tand T2: ";
      t2.Show();
}
```

该程序的输出为：

```
Original T1: X = 3Y = 4 Z = 5
After ++ t1, T1:   X = 4 Y = 5 Z = 6
      and T2:      X = 4 Y = 5 Z = 6
After t1 ++, T1:   X = 5 Y = 6 Z = 7
      and T2:      X = 4 Y = 5 Z = 6
```

在例 12.11 中，增量运算符或者返回一个类 Demo 对象的值（后置增量运算符重载），或者返回一个对类 Demo 对象的引用（前置增量运算符重载），这就方便了程序中行 A 和行 B 的赋值操作。若像例 12.10 中那样返回一个整型值，则无法将这样的值赋给一个对象。

注意程序中后置增量运算符重载中的函数体，由于这里对 3 个成员的增量运算分别用 3 条语句（而不是表达式）来完成的，因此使用前置与后置是没有任何区别的，而增量运算以前的值已存放在临时对象 temp 中，并且函数返回的正是该对象的值，从而达到的后置增量的目的。

12.2.2 下标运算符

下标运算符“[]”只能被重载为类的非静态成员函数，对该运算符的重载应具有以下的

一般形式：

type class_name :: operator [] (arg);

其中：arg 是函数的参数。

下标运算符重载必须带一个参数且仅能带一个参数，该参数相当于下标值，这是下标运算符原始定义所规定的。

例12.12 下标运算符重载。

```
//      EX12_12.CPP

#include      <iostream.h>

class   Demo {
            int     Array[5];
        public:
            Demo()      {}
            int&    operator [] (int i)     {return Array[i];}
};

void    main()
{
        Demo d;
        int     i;

        for(i = 0; i < 5; i ++)
            d[i] = i + 1;                   // A

        for(i = 0; i < 5; i ++)
            cout << d[i] << '\t';
        cout << endl;
}
```

该程序的输出为：

```
1     2     3     4     5
```

例 12.12 中，重载了的下标运算符返回对 int 类型的引用，因而可以将函数调用用做赋值运算符的左操作数，如程序中行 A 所示。

下标值通常总是一个非负整数，然而利用运算符重载则可以使用任何数据类型甚至一个对象来作为下标。

例12.13 一个工资表类。

```
//      EX12_13.H

#include      "ex12_8.h"

class   Salary_Tab    {
        private:
            Person          *Staffs;                    // 职工数组
            float   *Salary;                            // 与职工数组对应的工资数组
```

```
        int     nMax;                           // 工资表的长度
        int     nElem;                          // 工资表中已有元素的个数
    public:
        Salary_Tab(int = 50);                   // 构造函数，缺省表长为 50
        ~Salary_Tab()
        {
            delete []Staffs;
            delete Salary;
        }
        float   &operator [] (Person&);         // 下标运算符重载
        void    PrintTab();                     // 输出表中的内容
};


//      EX12_13A.CPP

#include    <iostream.h>
#include    <string.h>
#include    "ex12_13.h"

Salary_Tab :: Salary_Tab(int n)
{
    Staffs = new Person[n];
    Salary = new float[n];
    nMax = n;
    nElem = 0;
}

float   &Salary_Tab :: operator [] (Person &per)
{
    int     i;

    for(i = 0; i < nElem; i ++)                 // 在表中查找姓名为 per.Name 的元素
        if(strcmp((const char*)per, (const char*)Staffs[i]) == 0)
            return Salary[i];                   // 若找到则返回与其对应的工资
                // 循环结束则说明表中无对应无素，于是将 per 的信息插入到工资表中
    if(nElem == nMax)
        return Salary[nMax];                    // 若表已满，则返回最后一个表项的工资值

    Staffs[i] = per;                            // 将 per 插入表尾
    Salary[i] = 0;                              // 工资值暂时赋 0
    nElem ++;                                   // 元素个数加 1

    return Salary[i];       // 返回新插入表项的工资下标变量以供程序中为其赋值
}

void    Salary_Tab :: PrintTab()
{
    cout << "No.\t" << "Name\t" << "Age\t" << "Sex\t" << "Salary\n";

    for(int i = 0; i < nElem; i ++)
```

```
        cout << (i + 1) << '\t' << (const char*)Staffs[i] << '\t'
             << int(Staffs[i]) << '\t' << char(Staffs[i]) << '\t'
             << Salary[i] << endl;
}


//      EX12_13.CPP

#include    <iostream.h>
#include    <stdio.h>
#include    "ex12_13.h"

void    main()
{
    Salary_Tab    tab(20);
    Person        per;
    char    name[81], sex;
    int     age;
    float   sala;

    cout << "Please Enter datas...\n";
    while(1)      {                         // 循环输入职工信息及工资额
        cout << "Name: ";
        gets(name);
        if(strlen(name) == 0)
            break;                          // 若用户在输入姓名时仅输入了“回车”则退出循环
        cout << "Age: ";
        cin >> age;
        cout << "Sex: ";
        cin >> sex;
        cout << "Salary: ";
        cin >> sala;

        per.SetName(name);
        per.SetAge(age);
        per.SetSex(sex);
        tab[per] = sala;                    // 将职工信息插入表中并赋工资值
    }

    cout << "\nShow table...\n";
    tab.PrintTab();                         // 显示工资表

    cout << "Please enter a present staff's name: ";
    per.SetName(gets(name));                // 输入一个表中已存在的姓名并存入 per 中
    cout << tab[per] << endl;               // 输出对应的工资额
    cout << "Enter hew salary: ";
    cin >> sala;                            // 输入新的工资额
    tab[per] = sala;                        // 改变与姓名对应的工资额
    tab.PrintTab();                         // 显示修改了的工资表
}
```

该程序的功能在源程序中已注释清楚，请读者自行运行该程序并分析它的结果。

下标运算符重载的重要性在于它扩展了数组下标的概念，方便了程序中对各种数据类型的操作。比如，若将下标运算符的参数说明成一个字符型指针，而在调用该函数时的实参为一个文件名，这样就使得程序像使用普通变量那样对磁盘文件进行操作了。

12.2.3 其它几个特殊运算符的重载

还有几个运算符在重载时有一些特殊的要求，但是在实用中极少将它们重载。这里仅给出重载它们时的一般形式，及所受的限制，不再举例演示。

1. 函数调用运算符 函数调用运算符只能被重载成类的非静态成员函数，对该运算符的重载应具有以下的一般形式：

type class_name :: operator () (<arg_lisr);

与普通函数一样，函数调用运算可以带有零个或多个参数，但不得带有缺省的参数。

由于调用一个重载了的函数调用运算符时具有以下的形式：

```
class_name(<args>);
```

这很容易和构造函数搞混。而且，在一个类中完全可以利用一个普通的成员函数来实现函数调用运算符所执行的操作，同时还提高了程序的可读性。因此实用中几乎从不重载该运算符。

2. 成员选择运算符 成员选择运算符“.”不允许重载，而成员选择运算符“->”只允许被重载为类的非静态成员函数，并且不得带有任何参数。该运算符重载具有以下的一般形式：

class_name1 $ class_name :: operator -> ();

其中的“$”或者为*，表示函数的值为指向类 class_name1 的指针；或者为&，表示函数返回一个对类 class_name1 的引用；或者为空，表示函数的值为一个类 class_name1 的对象。

该运算符的重载基本没有什么实用价值。

3. new 和 delete 运算符 new 和 delete 运算符只能被重载为类的成员函数，而且，无论是否使用了关键字 static 进行修饰，它们均被重载为静态的成员函数。这两个运算符重载具有以下的一般形式：

void *class_name :: operator new(size_t, <arg_list>);
void *class_name :: operator delete(void*, <size_t>);

其中：size_t 为 C++语言预定义的一个数据类型：

```
typedef     unsigned        size_t;
```

即，它就是 unsigned int。上述定义大多数 C++软件包都是在系统头文件 STDDEF.H 中进行的，实际上许多具体实现中，系统头文件 ALLOC.H，MEM.H，STDIO.H，STDLIB.H 以及 STRING.H 中都有该定义。因此，在重载这两个运算符时必须利用#include 预处理指令至少将这些头文件中的一个嵌入到源程序文件中。

new 运算符重载中，可以利用可选的 arg_list 对分配到的动态内存进行初始化；而在 delete

运算符重载中，其可选取参数 size_t 却从不使用，因此在一个类中，delete 运算符不得重载一次以上。

我们知道，new 和 delete 运算符是用于动态分配内存的，而内存管理则属于操作系统的一个核心内容，内存使用不当有可能导致系统崩溃。因此，对操作系统的内存管理不熟悉的人最好不要重载这两个运算符，而对内存管理比较熟悉的人可能又不需要重载这两个运算符。

12.3 字符串类

人们对 C 语言批评最多的就是它对字符串的处理不像有些高级语言——比如，BASIC，Fortran 语言——那样方便。在 C 语言中，对字符串的处理均是通过软件包所提供的字符串处理库函数来实现的，对字符串的连接、比较等操作不像对数值量的加法、比较那样自然。而且根本就不存在对字符串的截取函数。

在 C++语言中，由于有了运算符重载这一有力的工具，可以通过定义一个字符串类，并在类中重载 C++预定义的运算符，使得对字符串的处理如同对数值量的处理一样方便自然。

12.3.1 定义字符串类

下面通过一个例子定义一个字符串类，读者可以在此基础上对其进一步地完善。

例12.14 字符串类。

```
//      EX12_15.H

#if !defined   _EX12_15_H_
#define        _EX12_15_H_

#include       <iostream.h>

class  String {
       private:
              char    *Str;                                    // 字符串首地址
              unsigned       Len;                              // 字符串的长度
              void    CopyString(char*, unsigned);
       public:
              String() : Str(0), Len(0)       {}               // 缺省构造函数
              String(const String&);
              String(char*);
              ~String()       {delete Str;}                    // 析构函数
              const char*     isIn(const char)        const;
              const char*     isSubStr(const char*) const;
              int      Length()       {return Len;}            // 取得字符串的长度
              String&         operator = (const String&);
              String&         operator -= (const String&);
              int      operator < (const String&)    const;
              int      operator == (const String&)     const;
```

```
        int     operator > (const String&)   const;
        int     operator ! ()   {return Str != 0;}                    // 取反运算符重载
        friend ostream&     operator << (ostream&, String&);
        friend istream&     operator >> (istream&, String&);
        friend String&      operator + (String&, const String&);
        friend String&      operator + (String&, char*);
};

#endif


//      EX12_15.CPP

#include      <string.h>
#include      "ex12_15.h"

void   String :: CopyString(char *s, unsigned l)
{
      Len = l;
      Str = new char[Len + 1];
      strcpy(Str, s);
}

String :: String(const String &s)
// 拷贝初始化构造函数
{
      CopyString(s.Str, s.Len);
}

String :: String(char *s)
// 用一个普通字符串来初始化的构造函数
{
      CopyString(s, strlen(s));
}

const char*   String :: isIn(const char c)   const
// 若串中存在参数 c 所代表的字符则返回该字符在串中第一次出现的位置，否则返回一个空指针
{
      char   *p = Str;

      while(*p)     {
            if(*p == c)
                  return p;
            p ++;
      }
      return 0;
}

const char*   String :: isSubStr(const char *s)     const
// 若 s 是字符串的子串则返回它在字符串中的首地址，否则返回一个空指针
{
```

```
        return strstr(Str, s);
}

String&         String :: operator = (const String &s)
// 赋值运算符重载
{
        delete Str;
        CopyString(s.Str, s.Len);

        return *this;
}

String&         String :: operator -= (const String &s2)
// "-="运算符重载
{
        char    temp[81], *p;

        strcpy(temp, Str);                              // 暂存源串
        if((p = strstr(temp, s2.Str)) != 0)     {       // 若 s2 是源串的子串，则
                delete Str;                             // 释放源串所占动态内存
                Len -= s2.Len;                          // 调整源串的长度
                Str = new char[Len + 1];                // 重新分配内存
                *p = '\0';                              // 将子串首字符置为字符串结束标志
                strcpy(Str, temp);                      // 复制前半段字符串
                p += s2.Len;                            // p 指向子串后边的那个字符
                strcat(Str, p);                         // 连接后半段
        }

        return *this;
}

int     String :: operator < (const String &s)         const
// 若 s 小于*this 则返回 1，否则返回 0
{
        if(strcmp(Str, s.Str) < 0)
                return 1;
        return 0;
}

int     String :: operator == (const String &s)     const
// 若 s 等于*this 则返回 1，否则返回 0
{
        if(strcmp(Str, s.Str) == 0)
                return 1;
        return 0;
}

int     String :: operator > (const String &s)          const
// 若 s 大于*this 则返回 1，否则返回 0
{
```

```
    if(strcmp(Str, s.Str) > 0)
        return 1;
    return 0;
}

ostream&    operator << (ostream &os, String &s)
// 插入运算符重载
{
    os << s.Str;                        // 输出字符串

    return os;
}

istream&    operator >> (istream &is, String &s)
// 提取运算符重载
{
    char    buff[81];

    is.get(buff, 81);                   // 输入一个不超过 80 个字符的字符串，可包含空白字符

    s.Len = strlen(buff);               // 获得所输入串的长度
    delete s.Str;                       // 释放动态内存
    s.Str = new char[s.Len + 1];        // 重新分配内存
    strcpy(s.Str, buff);                // 复制所输入的串

    return is;
}

String&     operator + (String &s1, const String &s2)
// “+”运算符重载，返回对象 s1 和 s2 的连接
{
    static String   temp;

    temp.Len = s2.Len + s1.Len;
    temp.Str = new char[temp.Len + 1];
    strcpy(temp.Str, s1.Str);
    strcat(temp.Str, s2.Str);

    return temp;
}

String&     operator + (String &s1, char *s2)
// “+”运算符重载，返回对象 s1 和普通字符串 s2 的连接
{
    static String   temp;

    temp.Len = s1.Len + strlen(s2);
    temp.Str = new char[temp.Len + 1];
    strcpy(temp.Str, s1.Str);
    strcat(temp.Str, s2);

    return temp;
}
```

程序中各成员均已注释清楚，这里仅对取反运算符重载做一补充说明。通常，利用 new 分配动态内存后，必须测试接受动态内存的指针是否为一个空指针，即判断内存分配是否失败。而本书因篇幅所限，大部分范例中均不做此测试。事实上，就本例而言，若做此测试并在内存分配失败后做出相应的处理，将会大大地增加程序的复杂程度。为此，本例中重载了取反运算符。若在程序中对某个类 String 的对象 str 有所怀疑，可以执行如下的操作：

```
if(!str)
    // 执行某种错误处理操作
```

另外，EX12_15.CPP 文件中对插入和提取运算符进行了重载，这一点将在第 13 章中详细介绍。

12.3.2 使用字符串类

有了以上的定义，在应用程序中就可以很方便地使用字符串了。

例12.15 字符串类的演示程序。

```
//      EX12_16.CPP

#include      <iostream.h>
#include      "ex12_15.h"

void   main()
{
       String  s1("The C++ language"), s2, s3;
       char    *sp = "++";

       cout << "S1:\t" << s1 << endl;

       cin >> s2;
       cout << "S2:\t" << s2 << endl;
       if(s1 > s2)
              cout << "S1 is large then S2\n";
       else   if(s1 < s2)
              cout << "S1 is less then S2\n";
       else
              cout << "S1 is equal to S2\n";

       if(s1.isIn('+'))
              cout << "The \'+\' is in the S1\n";
       if(s1.isSubStr(sp))    {
              cout << "SP is a sub-string of S1\n";
              s2 = s1 -= sp;
              cout << "After \"s2 = s1 -= sp\"\n";
       }
       else
              cout << "SP is not a sub-string of S1\n";
       cout << "S1:\t" << s1 << endl << "S2:\t" << s2 << endl;

       s3 = s1 + " " + s2;
```

```
    cout << "After \"s3 = s1 + \" \" + s2\"\n"
    cout << "S1:\t" << s1 << "\tS2:\t" << s2 <<endl;
    cout << s3 << "\tThe Length of S3 is " << s3.Length() << endl;
}
```

该程序的输出为：

```
S1:   The C++ language
The J++<CR>
S2:   The J++
S1 is less then S2
The '+' is in the S1
SP is a sub-string of S1
After "s2 = s1 -= sp"
S1:   The C language
S2:   The C language
After "s3 = s1 + " " + s2"
S1:   The C language   S2:   The C language
The C language The C language      The Length of S3 is 29
```

其中第二行为用户的输入。

请读者对照程序及其输出自行分析各语句的作用以及类 String 中各成员函数和友元函数的功能。

习题

12-1 根据你所具有的复数知识，定义一个比较完善的复数类。

12-2 为类 Person 及其派生类重载“==”和“!=”运算符。

12-3 为习题 9-8 中的类 Array 及其派生类（习题 10-2）重载各种运算符，尽可能使它们的对象操作起来既方便又自然。

12-4 为字符串类 String 重载“-”、“+=”和转换函数。

12-5 为字符串类添加 3 个成员函数：

```
char* String :: Left(int n);                  // 返回字符串中左起 n 个字符
char* String :: Right(int n);                 // 返回字符串中右起 n 个字符
char* String :: Middle(int pos, int n);       // 返回字符串中从第 pos 起的 n 个字符
```

12-6 将类 Person 及其派生类中的字符串型数据成员改写为类 String 的对象成员。

12-7 改写例 12.13 中的工资表类，将其数据成员 Staffs 和 Salary 归在一个类中：

```
class Item {
        Person      Staff;
        float  Salaary;
    public:
        //...
        friend class  Salary_Tab;
```

```
};

class Salary_Tab {
        Item  *Table;
        //...
    public:
        //...
};
```

并为这两个类定义相应的运算符重载和必要的转换函数。

第 13 章　C++语言的 I / O 流类

输入 / 输出（Input / Output，简称 I / O）是指程序与计算机的外部设备——比如：显示器、磁盘机、键盘等——交换信息的能力。一个算法语言 I / O 功能的强弱，是体现该语言整体功能强弱的一个重要指标。C++语言具有极强的 I / O 功能，本章介绍这方面的内容。

13.1　概述

与大多数高级语言不同，C 与 C++语言中没有独立的 I / O 语句。在 C 语言中，I / O 操作是通过调用 I / O 库函数来完成的，例如，在前些章中所使用的函数 getch（），gets（）等均属于 I / O 库函数；而在 C++语言中，除了可以像 C 语言那样调用 I / O 库函数外，它还提供了一个 I / O 流类体系。在实用中，程序员既可以使用前者也可以使用后者来完成所需的 I / O 操作。然而，C++语言只所以提供两套 I / O 操作手段，其目的主要是为了在输入输出方面提供对 OOP 的支持以及保持与 C 语言的兼容性。这里建议读者尽可能地使 C++语言的 I / O 流类来实现输入输出操作。本书对于 I / O 函数不做更多的介绍。另外，由于 C++的 I / O 流类系统比较复杂、成员函数数量庞大，本书不可能对它们进行全面地介绍。因此，有意对它们进行更深入地了解以及对 I / O 库函数有兴趣的读者可查阅有关手册。

13.1.1　流

C++语言的 I / O 系统为了向程序员提供一个统一的接口，使得程序的设计能够尽可能地与所访问的具体设备无关，在程序员与被使用的设备之间提供了一个抽象的概念——流（Stream）。

C++语言所能支持的外部设备是各种各样的，不同的外部设备的操作方式也是各不相同的。然而 C++的 I / O 系统将每个设备都转换成一个称为流的逻辑设备，由流来完成对不同设备的具体操作。而从程序员的角度看，所有流的行为都是相同的。因此，一个用来进行磁盘文件写入操作的函数，也可以用来向显示器写信息。

C++语言的流有两种类型：文本流（Text Stream）和二进制流（Binary Stream）。

文本流是一串 ASCII 字符。若一个磁盘文件上的数据是以文本流的形式存放的，则可以用 DOS 命令 TYPE 将其内容输出到显示器上，也可以利用各种文字处理器对其进行编辑。在不同的具体实现和不同的设备中，对文本流的处理方式不尽相同。比如，有的系统会将数据中的新行符“\n”转换成“\r\n”（回车换行）后才写入磁盘，而当将数据从磁盘中读出时又会做相反的处理。因此，一个以文本流形式存储的数据可能与其原始数据的字节数不相等。

二进制流是将数据以其二进制形式存放的，各种系统均不对二进制流中的数据进行任何

转换。一个以二进制流存储的数据与其原始字节数完全相等。

13.1.2 文件

流是C++语言对所有外部设备的逻辑抽象，而文件则是C++语言对具体设备的抽象。比如，一个磁盘文件是一个文件，而一台显示器也是一个文件。所有流的行为都是相同的，而不同的文件则可能具用不同的行为。比如，对磁盘文件既可以进行写操作也可以进行读操作；而对显示器文件则只能进行写操作、对键盘文件则只能进行读操作。

当程序与一个文件交换信息时，必须通过打开文件的操作将一个文件与一个流联系起来。一旦这个联系建立起来，以后对该流的访问就是对该文件的访问，也就是对一个具体设备的访问。如果一个文件支持随机访问，则打开文件时系统会自动将一个文件读或写指针（简称文件指针）置于文件的开头处。随着对文件的读写，系统将自动地调整该指针，使其指向下一个读写位置。

可以通过关闭文件的操作将一个文件与流的联系断开。对于以输出为目的而打开的文件，关闭它时系统将自动地把与该文件相关的缓冲区中的内容写入到文件中去，以保证文件内容的完整性。这一过程称为对这个流的“刷新”。当一个程序正常结束时，系统会自动地刷新所有的缓冲流并将所有打开的文件关闭。

13.1.3 缓冲

所谓缓冲，是指系统在内存中开辟一块区域用于临时存放信息。系统在内存中开辟的这一块存储区就叫做缓冲区。

一个流可以是缓冲的，也可以是非缓冲的。对于缓冲流，只有当缓冲区已满（称为溢出（Overflow））或者当前送入流中的数据为新行符时，系统才对流进行刷新。而对于非缓冲流，所有送入流中的数据将被立刻处理掉。由于外部设备的速度与CPU 相比往往要慢得多，频繁地与外设交换信息必将占用大量的机器时间，影响程序的运行速度。因此，多数流均为缓冲流。但有时为了某种特殊需要，也常使非缓冲流。

13.2 C++的基本流类体系

C++语言在头文件IOSTREAM.H中说明了一个基本流类体系，为C++程序进行I/O操作提供支持。因此，程序中如果需要利用流类进行 I/O 时，则必须使用嵌入预处理指令将该头文件插入到源程序中。

13.2.1 基本流类体系

C++语言的基本流类体系如图13-1所示。

从图13-1可以看出，类ios是所有基本流类的基类，其他的基本流类均从该类派生而来。图中类ios指向streambuf的双箭头在这里不是表示派生关系，它表示类ios中的一个指针成员，该指针指向类streambuf的一个对象。而类streambuf的作用是管理一个流的缓冲区。由

于数据隐藏和封装的关系，除非程序员需要派生自己的 I / O 类，一般不涉及该类。在实用中，只使用类 ios，istream 和 ostream 中所提供的公有接口进行 I / O 操作。类 ios 是类 istream 和 ostream 的虚基类，它提供了对流进行格式化 I / O 操作以及错误处理的方法。类 istream 和 ostream 分别从类 ios 中公有派生。前者提供对流进行提取（输入）操作的方法；而后者提供对流进行插入(输出)操作的方法。最后，类 iostream 通过多重派生，从类 istream 和 ostream 中共同公有派生。该类中除构造函数外并没有提供新的成员函数，它只是将类 istream 和 ostream 组合在一起，以支持对一个流进行双向（输入和输出）操作。

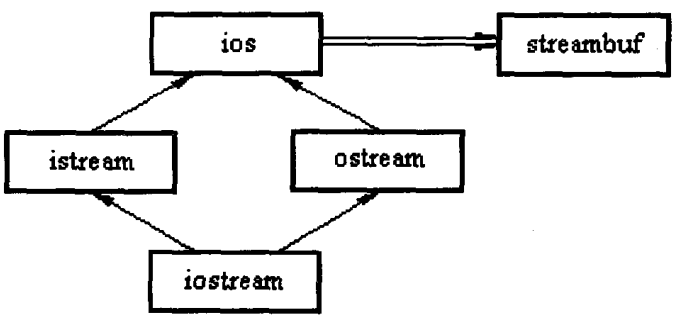

图 13-1　C++语言的基本流类体系

应当说明的是：不同的具体实现有时会对基本流类体系进行某种扩充，比如，在 Borland C++软件包中就又派生出 3 个用于提供预定义的所谓“标准”流的 withassign 类，它们分别从类 istream、ostream 和 iostream 公有派生而来：

```
class  istream_withassign : public istream;
class  ostream_withassign : public ostream;
class  iostream_withassign : public iostream;
```

13.2.2　预定义的流及流运算符

C++流类库预定义了 4 个流，它们分别为对应流类的对象：

```
extern istream     cin;
extern ostream     cout;
extern ostream     cerr;
extern ostream     clog;
```

在 Borland C++中，它们则分别是对应 withassign 类的对象：

```
extern istream_withassign     cin;
extern ostream_withassign     cout;
extern ostream_withassign     cerr;
extern ostream_withassign     clog;
```

这 4 个流称为“标准”流，这是因为每当应用程序中嵌入头文件 IOSTREAM.H 时，它们的构造函数就被调用。正因为如此，只要应用程序中嵌入了 IOSTREAM.H，则在程序运行时这 4 个对象（流）就自然存在。

流 cin 称为标准输入流，它缺省地与标准控制台输入文件（一般为键盘）相联系；流 cout 称为标准输出流；流 cerr 和 clog 称为标准错误输出流，3 个标准输出流缺省地与标准控制台输出文件（一般为显示器）相联系。这 4 个流中，cerr 为非缓冲流，而其余 3 个标准流均为缓冲流。

C++预定义的流运算符为插入运算符“<< ”（按位左移运算符的重载）和提取运算符“>>”（按位右移运算符的重载）。前者用于输入流，而后者用于输出流。

在对标准流进行 I / O 操作时，系统将自动地对进入流的数据进行转换。即，将所有送入输出流中的数据都转换成字符串的形式后再输出到文件中；将所有送入输入流中的字符串形式的数据都转换成与接收该数据的变量同类型的二进制形式后再赋给变量。

例 13.1 使用 C++预定义的流。

```
//      EX13_1.CPP

#include      <iostream.h>

int     main(int argc, char *argv[])
{
        if(argc < 2)    {
                cout << "USAGE: EX13_1 someStrings<CR>" << endl;          // A

                return 1;
        }

        cout << "Please enter an integer: ";                              // B

        int   num;

        cin >> num;                                                       // C

        for(int i = 1; i < argc; i ++)
                cout << argv[i] << ' ';
        cout << endl;

        cout << "The number you entered is " << num << endl;

        return 0;
}
```

例 13.1 中使用了带参数的 main（）函数，因此运行该程序时除了键入程序名外还需键入一些字符串。若运行该程序时键入：

ex13_1<CR>

则程序将显示：

USAGE: EX13_1 someStrings<CR>

并退出运行。

若运行时键入：

ex13_1 This is a test<CR>

则程序将输出：

Please enter an integer:

并等待用户输出一个整数。设用户对此信息的响应为键入一个 35 并按下回车键，则程序将显示以下信息并结束运行：

This is a test
The number you entered is 35

许多操作系统——如 DOS，UNIX——都提供有一种所谓的 I / O 重定向技术，该技术可以使用户在运行一个程序时将系统所定义的标准输入或输出设备指定为其他的设备。例如，在 DOS 下，可以采用键入如下命令的方式将程序 EX13_1 中输出到标准输出流 cout 中的数据送到磁盘文件 FILE.DAT 中去：

ex13_1 This is a test > file.dat<CR>

其中，符号“>”为 DOS 输出重定向符。

经过这样的重定向，程序中所有对流 cout 的插入操作都将输出到磁盘文件 FILE.DAT 中，而不再显示在屏幕上。如果在运行该程序时忘记了键入字符串，仅输入了程序名：

ex13_1 > file.dat<CR>

则由于程序中行 A 的提示信息被送到磁盘文件中去，而不再在屏幕上显示，用户将会感到程序莫名奇妙地结束了。更为严重的是，即就是正确地键入了字符串，由于程序中行 B 的提示信息不再显示，更是给人一种死机的假象，因为用户可能不知道程序正在等待他输入一个整数。

为解决这样的问题，可将程序中行 A 与行 B 对流 cout 的插入改为对流 cerr 插入操作。因为 C++语言规定：标准错误输出流不得重定向，它们所连接的输出设备永远是显示器。经过这样的改动，这两行的提示信息在需要时就会输出到显示器上而不会被输出到磁盘文件中去了。

标准流 clog 与 cerr 一样，不能被重定向。那么将上述的改变用 clog 来代替，结果将如何呢？从 13.1.3 中的介绍可知，当缓冲区未满且没遇到新行符时，对缓冲流的插入操作并不立即产生输出。因此，若将程序中行 A 与行 B 的 cout 改为 clog，则无论是否重定向，用户总是不能事先看到行 B 的提示，也会造成一种死机的假象。这是由于行 B 中欲输出的信息尚不足以导致输出缓冲区溢出，且该信息中无新行符之故。若用户在看不到提示的情况下输入一个整数，程序才会显示行 B 的提示信息。这是因为 C++语言在遇到对输入流的提取操作时，会自动刷新输出流。当用户输入一个整数后，程序中行 C 的 cin >> num 得以执行。这时，系统首先刷新输出流，然后才执行对输入流的提取。同样，对于缓冲输入流，系统遇到对输出流的插入时也会刷新它的。

13.2.3 流的格式化 I / O

所谓格式化 I / O 是指按指定的格式对流进行 I / O 操作。比如，按十六进制输入一个整数、规定输出数据的宽度等。一般情况下，格式化 I / O 总是针对文本流的。

在对流进行 I / O 操作时，难免会出现一些错误。一个完善的 I / O 系统应能提供发现错误及改正错误的手段。

C++语言在类 ios 中说明了对流进行格式控制与错误处理所需的方法。

13.2.3.1 格式控制符

C++在类 ios 中说明了一个用于控制 I / O 格式的公有成员，它是一个无名枚举成员：

```
enum {
    skipws      = 0x0001,      // 跳过输入中的空白字符。I
    left        = 0x0002,      // 输出左对齐。O
    right       = 0x0004,      // 输出右对齐。O
    internal    = 0x0008,      // 在符号或数制字符后进行填充。O
    dec         = 0x0010,      // 将数据转换成十进制。I / O
    oct         = 0x0020,      // 将数据转换成八进制。I / O
    hex         = 0x0040,      // 将数据转换成十六进制。I / O
    showbase    = 0x0080,      // 输出中带有一个表示数制的字符。O
    showpoint   = 0x0100,      // 输出实数时必须带有小数点。O
    uppercase   = 0x0200,      // 十六进制数据以大写形式输出。O
    showpos     = 0x0400,      // 大于零的数据前应带“+”号。O
    scientific  = 0x0800,      // 以指数形式输出实数。O
    fixed       = 0x1000,      // 以定点（一般）形式输出实数。O
    unitbuf     = 0x2000,      // 在插入后立即刷新流。O
    stdio       = 0x4000       // 在插入后立即刷新 stdout 和 stderr。O
};
```

其中：注释中的字母 I 表示该控制常量适用于输入；字母 O 表示该控制常量适用于输出；而 I / O 则表示该控制常量既适用于输入又适用于输出；stdout 和 stderr 分别对应标准输出设备和标准错误设备。

这些格式控制常量虽然是定义成枚举型，然而，由于每个枚举成员的取值形式是按位取值的，所以可以用一个 long 型量来存放它们的组合。实际上，C++语言正是这样做的，它用一个称为标志字的 long 型变量来存放对 I / O 格式的设置，该标志字为类 ios 的保护成员：

```
protected:
    long  x_flags;
```

成员 ios :: x_flags 的值从右向左每一位对应一个格式控制符。因此，可以利用位操作来设置或取消一个格式控制符。

应当说明的是：不同的具体实现中各格式控制常量的值不一定相同，但各控制常量名则是标准中所规定的。因此，在实用中应尽量避免使用控制常量的值。

13.2.3.2 格式控制函数

应用程序可以通过类 ios 中的成员函数 ios :: flags（），ios :: setf（）和 ios :: unsetf（）来取得、设置和取消标志字中的指定位。

在类 ios 中，重载了两个取得格式控制标志字的成员函数 flags（）：

```
long  ios :: flags();
long  ios :: flags(long);
```

其中：函数 flags（）仅返回当前标志字的值；而函数 flags（long）则不仅返回当前标志字的值，而且其参数还用来设置以后的 I / O 格式。

例 13.2 使用函数 flags（）取得和设置标志字。

```
//      EX13_2.CPP

#include      <iostream.h>

void   main()
{
       cout << "Enter an integer: ";

       int   num;

       cin >> num;
       cout << "Dec: " << num << '\t';                              // A
       cout.flags(ios :: showbase | ios :: hex | ios :: uppercase);  // B
       cout << "Hex: " << num << '\t';
       cout.flags((cout.flags() & ~ios :: hex) | ios :: oct);        // C
       cout << "Oct: " << num << endl;
}
```

设用户运行该程序时对其响应如下：

Enter an integer: **27**<CR>

其中，27 前边的空白表示用户按下了一些空白字符键。这时，程序的输出为

Dec: 27 Hex: 0X1B Oct: 033

由于 C++语言缺省的输入格式为跳过空白字符、数制为十进制，因此在所输入的数据前边尽管有一些空白字符，标准输入流仍能正确地提取输入数据（注意：这里的空白字符不包括“回车”符）。由于 C++语言缺省的输出格式为十进制，所以对行 A 的插入操作不需要进行格式设置。程序中行 B 将标准输出流的输出格式设置成十六进制，带表示数制的字符，字母大写，其效果可以从输出中看出。行 C 首先取得当前标志字，然后将该标志字同表示十六进制数位的反进行按位与运算，即取消十六进制输出，最后设置输出为八进制。由于这里没有取消数制字符，所以输出带有八进制数制字符“0”。

应当说明的是：行 C 中取消十六进制格式的操作是非常必要的，因为有的 C++语言具体实现是从左向右测试标志字的，而在表示 3 种数制的控制符中，表示十六进制的位居最左。因此，若标志字中该位未复位，则设置的十进制或八进制均不起作用。这就是说，若在行 C 中仅仅设置八进制，则程序中最后一行的输出可能仍为十六进制形式。

调用 flags（）函数时，其参数也可以写成常量形式。比如，行 B 可以改写为

```
cout.flags(0x0080 | 0x0040 | 0x0200);
```

但是，由于 13.2.3.1 中所说明的理由，这样的用法移植性较差。

从格式控制符的说明可以看出，数制的设置也可以作用于输入流。因此，若在例 13.2 中将输入流 cin 也设置为十六进制，则只有输入的数据为 1b 或 1B 时输出才与上述相同。在此情况下，输入数据时不需要键入十六进制前缀“0x”。

在类 ios 中，亦重载了两个设置标志字的成员函数 setf（）：

```
long  ios :: setf(long);
long  ios :: setf(long _setbit, long _field);
```

其中：函数 setf（long）中的参数即为欲设置之标志字值；函数 setf（long, long）的_setbit 参数为所设置的格式控制符之值；而_field 参数则指出欲设置的是标志字中的哪一位。这两个函数均返回以前标志字的值。

在类 ios 中还说明了一个用于清除标志字中某位的成员函数 unsetf（）：

```
long  unsetf(long);
```

该函数清除标志字中由参数所指定的位，同时返回以前的标志字。

例 13.3 使用函数 setf（）设置标志字。

```
//      EX13_3.CPP

#include      <iostream.h>

void   main()
{
       cout << "Enter an integer: ";

       int   num;

       cin >> num;
       cout << "Dec: " << num << '\t';
       cout.setf(0x02c0);                                  // A
       cout << "Hex: " << num << '\t';
       cout.unsetf(ios :: hex);                            // B
       cout.setf(ios :: oct, ios :: oct);                  // C
       cout << "Oct: " << num << endl;
}
```

若对该程序的响应与例 13.2 相同，则其输出也与例 13.2 完全一样。

程序中行 A 直接用一个数值常量来设置标志字，它相当于语句：

```
cout.setf(ios :: hex | ios :: showbase | ios :: uppercase);
```

行 B 用来取消十六进制输出；行 C 设置八进制输出格式，其中参数也可以写成常量形式：

```
cout.setf(0x0020, 0x0020);
```

标志字中有些位是相互矛盾的，比如 dec，oct 和 hex；left，right 和 internal；scientific 和 fixed。这些位中，任何时刻对应位仅有一位允许置位，其余位必须复位。这势必给编程带来不便，这一点从以上两例中就可以看出。为解决这一矛盾，方便使用，C++语言在类 ios

中说明了 3 个静态成员用于作为函数 setf（）的第二个参数：

```
static const long   basefield;      // 其值为 dec | oct | hex
static const long   adjustfield;    // 其值为 left | right | internal
static const long   flaotfield;     // 其值为 scientific | fixed
```

利用这些静态成员，就可以很方便地设置标志字中的某些位。比如，无论当前的数制为何，均可以用以下的函数调用将输出设置成八进制，而无须清除原数制设置：

```
cout.setf(ios :: oct, ios :: basefield);
```

在源程序 13_3.CPP 中，可以用这一行来代替其中的行 B 和行 C 两条语句。

13.2.3.3 设置域宽与填充方式

设置域宽是指为文本流的 I / O 指定一个输入输出项所占的字符个数。类 ios 中重载了两个用于设置域宽的成员函数：

```
int    ios :: width();
int    ios :: width(int);
```

前者仅返回当前的域宽；而后者可用来设置下一次 I / O 的域宽并返回以前的域宽。

设置填充方式是指用一个指定的字符用来填充输出域中的空位。类 ios 中重载了两个设置填充方式的成员函数：

```
char   ios :: fill();
char   ios :: fill(char);
```

前者仅返回当前的填充字符；而后者可用来设置以后的填充字符并返回以前的填充字符。

例 13.4 域宽和填充。

```
//      EX13_4.CPP

#include      <iostream.h>

void   main()
{
       int      num = -23, len;

       cout << "12345678901234567890\n";
       len = cout.width();                          // 取得缺省的域宽
       cout << num << '\t' << len << endl;
       cout.width(10);                              // 将域宽设置为 10
       len = cout.width();                          // 取得当前域宽
       cout.fill('#');                              // 将填充字符设置为'#'
       cout << num << '\t' << len << endl;
       cout.flags(ios :: left);                     // 设置输出左对齐
       cout.width(len);                             // 恢复域宽为 10
       cout << num << '\t' << len << endl;
       len = cout.width(10);                        // 设置并取得以前的域宽
```

```
    cout.flags(ios :: internal);                    // 设置填充方式为中间填充
    cout << num << '\t' << len << endl;
}
```

该程序的输出为：

```
12345678901234567890
-23       0
#######-23    10
-23#######          10
-#######23          0
```

请读者自行根据源程序中的注释并对照程序的输出分析各种设置的作用。

应当说明的是：函数 width（long）仅对其后最近的一次 I / O 操作起作用，然后立即将域宽恢复为缺省状态。例如，程序中行 A 将输出域宽设置成 10 个字符，而在行 B 中，当向显示器插入变量 num 的值后，域宽就恢复为缺省值 0。同理，行 C 中取得的域宽值自然为 0。

13.2.3.4 实数输出格式控制

在类 ios 中，除了说明有用于控制实数输出形式的格式控制符 ios :: scientific 和 ios :: fixed 外，还重载了两个用于设置实数精度的成员函数 precision（）：

```
int    ios :: precision();
int    ios :: precision(int);
```

前者仅返回当前的实数精度设置值；后者设置以后的实数输出精度并返回以前的精度值。实数的输出精度用一整型数据表示输出时其小数的位数，该值缺省为 6。

例 13.5 实数输出格式控制。

```
//      EX13_5.CPP

#include      <iostream.h>

void   main()
{
      float   num = -123.4567;

      cout << num << endl;
      cout.setf(ios :: scientific, ios :: floatfield);
      cout << num << endl;
      cout.unsetf(ios :: scientific);
      cout.width(15);
      cout.precision(2);
      cout << num << endl;
}
```

该程序的输出为：

```
-123.456703
-1.234567e+02
        -123.46
```

从例 13.5 的输出可以看出，对实数输出时系统还对数据进行四舍五入调整。

13.2.3.5 流的刷新

在解释例 13.1 时曾指出：若将提示信息送 clog 输出，则由于该流为一缓冲流而使用户不能及时看到提示信息。为解决这一问题，C++ 提供了强制刷新一个流的手段。其方法之一就是设置格式标志字中的 unitbuf 或 stdio 位。

例 13.6 刷新输出流。

```
//      EX13_6.CPP

#include      <iostream.h>

int     main(int argc, char *argv[])
{
        if(argc < 2)    {
                cerr << "USAGE: EX13_1 someStrings<CR>" << endl;

                return 1;
        }

        clog.setf(ios :: unitbuf);                      // A
        clog << "Please enter an integer: ";            // B

        int   num;

        cin >> num;

        for(int i = 1; i < argc; i ++)
                cout << argv[i] << ' ';
        cout << endl;

        cout << "The number you entered is " << num << endl;

        return 0;
}
```

无论是否进行了重定向，例 13.6 均能正确及时地在显示器上输出提示信息。这是由于程序中行 A 将标志字中的 unitbuf 位置位，这就使得行 B 中对 clog 的插入操作引起对该流的刷新，致使该行一经执行便立即产生输出。

13.2.3.6 I/O 操纵符

使用类 ios 的成员函数进行格式控制时，每一操作都要调用一次函数。这不仅使得源程序书写起来不方便，而且程序运行时的时间开销也较大。为改善这一情况，C++ 为格式控制提供了 13 个操纵符（Manipulator），如表 13.1 所示。

表 13-1 中：setbase（）的参数取值为 0，8，10 或 16。0 为缺省数制，通常表示十进制。

这些操纵符均在头文件 IOMANIP.H 中说明，因此，为使用这些操纵符，必须在源程序中嵌入该头文件。

表 13.1　C++语言的 I / O 操纵符

操纵符	意义	输入 / 输出
dec	将数据转换成十进制	I / O
hex	将数据转换成十六进制	I / O
oct	将数据转换成八进制	I / O
setbase(int)	将数据转换成由参数所指定的数制	I / O
ws	提取空白字符	I
ends	插入一个表示字符串结束的 nul 字符	O
endl	插入一个新行符	O
flush	刷新流	O
resetioflags(long)	清除指定标志位	I / O
setioflags(long)	设置指定标志位	I / O
setfill(int)	设置填充字符	O
setprecision(int)	设置实数的精度	I / O
setw(int)	设置域宽	O

例 13.7　使用操纵符。

```
//      EX13_7.CPP

#include     <iostream.h>
#include     <iomanip.h>

void   main()
{
       int     num = -23, len = 0;

       cout << "12345678901234567890\n";
       cout << num << '\t' << len << endl;
       len = 10;
       cout << setw(len) << setfill('#') << num << '\t' << len << endl;
       cout << setiosflags(ios :: left) << setw(len) << num << '\t' << len << endl;
       cout << setiosflags(ios :: internal) << setw(len) << num << '\t' << len << endl;
}
```

该例实际上是例 13.4 的另一版本，因此，其输出与例 13.4 基本相同。所不同的是本例输出中的最后一个数据为 10，而不是 0。其中原因请读者自行分析。

应当说明的是：各操纵符与对应的标志字位的作用完全相同。比如，与函数 width（）一样，操纵符 setw（）也仅对其后的一次有效插入起作用；hex 的作用也延续到下一次数制设置等。

13.2.4　流的错误处理

在 I / O 过程中，难免会发生操作错误。比如，输入一个十进制数时中间夹有字母；向磁盘文件中写信息时磁盘已满，等等。这时，系统应当能发现错误并尽可能地纠正错误。C++语言的流类具有发现错误的功能，并且，一旦发现 I / O 操作出错，就将错误的性质记录下来以便用户利用程序进行检测。

C++语言在类 ios 中说明了一个名为 io_state 的公有枚举成员，其中定义了 5 种错误状态：

```
enum      io_state    {
      goodbit     = 0x00,        //I / O 操作正常
      eofbit      = 0x01,        // 已达文件尾，输入流中已无数据可提取
      failbit     = 0x20,        //I / O 操作出错
      batbit      = 0x40,        // 非法 I / O 操作
      hardfail    = 0x80         // 不可恢复的严重错误
}
```

从 ios :: io_state 的定义可以看出，预定义的流的错误状态也是以位方式定义的，可以用一个整型变量来存放这些状态。事实上，C++语言也正是这样做的。类 ios 中还说明了一个整型保护成员 state，该成员的低四位用来记录最近一次 I / O 操作的错误状态。若 state 的值为 0（即，goodbit），则说明操作正常，无错误发生；若 state 的某位被置位，则说明操作出现了对应位所表明的错误。

前两种错误状态的意思已经很明确，这里只对后 3 种错误状态作进一步的说明。failbit 表示 I / O 操作中出现的错误，比如，欲提取一个数值量时而流中却是一个字符；badbit 表示 I / O 操作是非法的，比如，欲从一个只写的文件中读取数据；而 hardfail 则表示 I / O 操作中出现了严重错误，比如，向一个磁盘文件写数据时磁盘已满。由 failbit 和 badbit 所标识错误为可恢复的错误，即，若发生这样的错误，只要清除流的相应错误状态位后，仍可正常地进行 I / O 操作，因为流中的信息没有受到破坏。而 hardfail 所表示的错误则属严重错误，一般情况下，这时流中的信息已遭破坏，因而是不可恢复的。

流的错误状态，即成员 state 的值，由系统根据最近一次 I / O 操作的结果自动置位，用户一般不用考虑对该成员的赋值。而 C++语言在类 ios 中说明了几个成员函数，用来读取错误状态，以使用户程序可以随时了解 I / O 操作是否正常，以便做出相应的处理：

```
int    ios :: rdstate();          // 读取 state 的值
int    ios :: good();             // 若 state = 0 则返回非零值
int    ios :: eof();              // 若 eofbit 位置位则返回非零值
int    ios :: fail();             // 若 failbit，badbit 或 hardfail 位置位则返回非零值
int    ios :: bad();              // 若 badbit 或 hardfail 位置位则返回非零值
void   ios :: clear(int i = 0);   // 清除指定错误位
```

成员函数 clear（）是一个带有缺省参数的函数，其参数的缺省值为 0。该函数的主要目的是用来清除流的错误状态，以便程序可以对发生错误的流继续进行 I / O 操作。然而，若不使用缺省参数，则可以利用该函数对指定流的 state 成员进行置位。比如语句：

```
cin.clear(ios :: bedbit);
```

就将标准流 cin 的 state 成员值强制设置成 ios :: bedbit 值。而语句：

```
cout.clear(cout.rdstate() | ios :: eofbit);
```

就将标准流 cout 的 state 成员中与 eofbit 对应的位置位且不改变其他位的值。

注意：函数 clear（）不能清除由 hardfail 导致的错误。

利用这些成员函数，用户程序可以检测一次 I / O 操作是否发生了错误。若发生了可恢复的错误，用户程序还可以利用函数 clear（）来清除错误，以便对发生错误的流能继续进行 I / O 操作以及使程序能正常地继续运行。

例 13.8 流的错误处理。

```
//      EX13_8.CPP

#include      <iostream.h>

void   main()
{
       int     i, s;
       char    str[81];

       cin >> i;
       s = cin.rdstate();                    // A
       while(s)       {
              cin.clear();
              cin >> str;
              cout << "Input error, try again: ";
              cin >> i;
              s = cin.rdstate();
       }

       cout << i << endl;
}
```

程序中行 A 取得标准输入流的错误状态，并将所获得的错误状态值赋给整型变量 s。若 s 的值不为 0，则说明输入有错，从而导致 while 循环的执行。由于 C++语言中没有提供刷新标准输入流的手段，因此，在循环中首先清除输入流的错误状态，然后将所输入的数据读入一个字符串中。接着，输出一个提示信息要求用户重新输入。最后，接收用户的输入并检测流的错误状态。这一过程直到输入正确为止。这里之所以用一个字符串来空读输入，是因为对于标准输入流 cin 来讲，无论什么样的输入数据对字符串来说都是合法数据。

由于 C++在类 ios 中重载了取反运算符“!”：

```
int    ios :: operator ! () {return fail();}
```

因此，可以利用该运算符重载使得例 13.8 变得更加简炼。

例 13.9 例 13.8 的另一版本。

```
//      EX13_9.CPP

#include      <iostream.h>

void   main()
{
       int     i;
       char    str[81];
```

```
    while(!(cin >> i))     {                    // A
        cin.clear();
        cin >> str;
        cout << "Input error, try again: ";
    }

    cout << i << endl;
}
```

该程序在行 A 调用了对象 cin 中重载了的取反运算符，若该函数调用的结果为真，则说明对象的 state 成员中对应的 failbit，badbit 或 hardfail 的位被置位，从而导致循环体的执行。

在第 2 章中曾提醒过读者：用户必须保证对 cin 的响应的正确性，否则其后果将是不可预知的。这是因为若某次提取操作出错，由于错误状态没有清除，其后的提取操作都将出错的原故。应当说明的是，在以上两例中，若遇到 hardfail 位置位时，将会变成死循环。所幸的是，对标准输入流来讲，这种情况一般是不会发生的。

有了这里的错误处理手段，就可以写出适应性和容错能力很强的程序。

13.3 输入和输出

C++语言中的类 istream 提供了创建输入流以及从流中输入数据的手段；类 ostream 提供了创建输出流以及向流中输出数据的手段；而类 iostream 则同时提供了上述两种手段。但由于类 iostream 的主要作用是用来派生其他可同时进行 I / O 操作的流类，在格式化 I / O 操作中一般不涉及该类。

对于格式化 I / O，类 istream 和 ostream 通过对运算符“>>”和“<< ”的多次重载，提供了 3 大类 13 种数据类型的提取和插入操作。这 3 大类数据类型为：字符型，包括 signed char 和 unsigned char；字符串型，包括 signed char*和 unsigned char*；数值型：包括 short，unsigned short，int，unsigned，long，unsigned long，float，double 和 long double。这对用户程序进行 I / O 操作提供了极大的方便。

13.3.1 格式化输入

格式化输入是通过对类 istream 的对象，利用重载了的运算符“>> ”进行提取操作来实现的。

最常用的格式化输入是对标准流 cin 的提取。程序员也可以定义自己的输入流对象。然而，由于定义用户流时将涉及到对流缓冲区 streambuf 的操作，且定义用户格式化 I / O 流的实用价值不大，因此本书对此不做进一步的介绍。

格式化输入缺省的格式为十进制、跳过前导空白字符。因此，对语句：

```
cin >> i;
```

的响应可以是任何空白字符（包括回车符）后跟一个十进制数（这里假设 i 是一个整型变量）形式的字符串。流 cin 将跳过输入串中的前导空白字符，并把输入串转换成一个整数值赋给变量 i。

格式化输入流是缓冲流，用户键入的字符存入流中一个指针所指的缓冲区中，只有在用户键入有效字符又按下了回车键（在有效字符之前键入的回车键将视为空白字符）后，系统才对输入流进行提取操作。若在提取数值型数据时，缓冲区中第一个字符为非数字字符，则该流的错误状态字中的 failbit 位被置位。除非将错误状态字复位为 0，否则以后对该流的提取操作均不能正常进行。输入缓冲区里有效字符序列中的空白字符以及与所提取数据类型不匹配的字符（称作非法字符。比如，提取数值量时遇到了非数值字符）均导致一次提取操作的结束，且该空白字符或非法字符仍留在输入缓冲区中。例如，若对上述输入的响应为

12r3<CR>

则系统将从标准输入流中提取字符 '1' 和 '2'，并把它们转换成整数 12 后赋给整型变量 i；而非法字符 'r' 及其后边的字符 '3' 仍留在 cin 的缓冲区中。这将导致以后对 cin 提取的失败。

若输入串中的第一个有效字符就是一个非法字符，则本次提取操作失败。比如，若对上述输入的响应为

r123<CR>

则键入的所有字符都留在输入缓冲区中，变量 i 没有被赋值。

由于对输入流进行提取操作时，未提取的字符仍留在缓冲区中。则遇到下一次对同一个流进行提取操作时，系统将从留在缓冲区中的第一个字符开始提取。于是，对以下的语句序列：

```
cin >> i;
//...
cin >> j;
```

则输入

12 34<CR>

将导致两条语句都得到执行。也就是说，当执行到第二条语句时，系统将不再等待用户键入数据，而是直接将缓冲区中的 34 读给变量 j。

对于链式输入：

```
cin >> i >> j >> k;
```

响应

1 2 3<CR>

和

1<CR>
2<CR>
3<CR>

都可以将整数 1，2，3 分别赋给变量 i，j，k。这是由于回车符作为空白字符导致一次提取操作的结束，而它本身还留在输入缓冲区中成为下一次提取时的前导空白字符之故。

13.3.2 输入操作函数

虽然提取运算符经过多次重载使得用户可以很方便地输入各种基本数据类型，然而，由于预定义的提取操作在遇到空白字符时就结束，这就使得需要输入空白字符或包含有空格的字符串时遇到了困难。当然，这一问题可以像前几章中那样，利用系统的库函数 getch（），getchar（），gets（）来解决。但是，这些函数属于 C 语言的库函数，C++中主要是为保持与 C 语言的兼容性才保留这些函数的。为了保持语言的统一性和完整性，C++语言在类 istream 中也说明了几个用于提取字符和字符串的非格式化成员函数。同时，为了简化接口，这些成员函数采用了重载的形式：

```
int     istream :: get();
istream&    istream :: get(signed char&);                    // A
istream&    istream :: get(unsigned char&);                  // B
istream&    istream :: get(signed char*, int, char = '\n');  // C
istream&    istream :: get(unsigned char*, int, char = '\n'); // D
```

在实用中，函数 A 和 B；C 和 D 实际上没有什么本质上的区别。因此，可以将这些成员函数看作是 3 个函数 get（），get（char&）和 get（char *buf, int n, char end = '\n'）。

除了函数 get（）外，其余的重载函数均返回对 istream 的引用，这是对输入函数重载的标准形式。而函数 get（）只所以返回一个整型值，其目的是为了能够提取文本流的结束标志 EOF。宏 EOF 在头文件 IOSTREAM.H 中被定义为-1，它只对文本流有定义。

所有 5 个 get（）函数均可以从输入流中提取空白字符，包括回车符。函数 C 和 D 从输入流中提取最多为 n - 1 个字符，存入字符串 buf 中，遇到参数 end 所给出的表示结束的字符或已提取了 n - 1 个字符则结束提取操作，并向 buf 中放入一个字符串结束标志。参数 end 的缺省值为回车符。若用户键入的字符多于 n - 1 个，则多余的字符将留在流的缓冲区中。使用这两个函数时，程序员应保证 buf 的实参所指缓冲区的大小不小于 n。

例 13.10 使用的 get（）函数。

```
//      EX13_10.CPP

#include      <iostream.h>

void    main()
{
        char    ch, str[81];

        cin.get(str, 81);                       // A
        cout << str << endl;
        while(cin.get() != '\n');               // B
        cin.get(ch);                            // C
        cout << ch << endl;
}
```

本例中，行 A 从标准输入流向字符数组 str 中提取最多 80 个字符，遇到回车符结束提取操作。行 B 的目的是为了从输入流的缓冲区中提取掉多余的字符和回车符。这一行是非常

必要的。因为若无行 B，则无论用户键入的字符是否超过 80 个，cin 的缓冲区中至少还留有一个回车符。这将导致行 C 不待用户键入任何字符便提取完毕。当然提取的不会是用户所期望的字符。若将提取字符串与提取字符的顺序倒过来，也存在这样的问题。

例 13.11 例 13.10 的另一种形式。

```
//      EX13_11.CPP

#include      <iostream.h>

void   main()
{
       char    ch, str[5];

       cin.get(ch);
       cout << ch << endl;
       cin.get();
       cin.get(str, 5);
       cout << str << endl;
}
```

该例的执行过程请读者自行分析。

为了方便字符串的输入，类 istream 中还说明了几个主要用于提取字符串的非格式化成员函数：

```
istream&    istream :: getline(signed char*, int, char = '\n');
istream&    istream :: getline(unsigned char*, int, char = '\n');
istream&    istream :: ignore(int = 1, int = EOF);
int     istream :: gcount();
```

其中：第一、二两个函数在实用中没有什么本质区别，它们的参数的意义及用法与 get（）函数中 C 和 D 相同，不同的是 getline（） 函数在所提取的字符个数没有超出指定值时，还将输入缓冲区中表示结束的字符也提取掉，但不存储它；第三个函数从流缓冲区中提取其第一个参数所指定个数的字符，其第二个参数给出提取结束字符。该函数在提取到规定个数的字符或提取到结束字符后结束操作，但不存储所提取的字符；第四个函数返回最近一次提取的字符个数。

例 13.12 使用函数 getline（）和 gcount（）。

```
//      EX13_12.CPP

#include      <iostream.h>

void   main()
{
       char    ch, str[81];

       cin.getline(str, 81);                    // A
       cout << cin.gcount() << endl;            // B
       cin.get(ch);                             // C
```

```
    cout << str << endl;
    cout << ch << endl;
}
```

本例行 A 中使用缺省的结束字符 '\n' 调用函数 getline（）来输入一个字符串。由于该函数提取流缓冲区中的结束字符，因此，在执行行 C 的字符提取操作之前不用清缓冲区。行 B 用来输出行 A 实际提取的字符个数（包括结束字符）。

若为函数 getline（）指定了一个非缺省的结字符，则回车符也可以作为字符串的有效字符。这一点也适用于对应的 get（）函数。

例 13.13 指定一个非缺省的结束字符。

```
//      EX13_13.CPP

#include      <iostream.h>

void   main()
{
      char    ch, str[81];

      cin.getline(str, 81, '#');              // A
      cout << cin.gcount() << endl;
      cin.ignore(80, '\n');                   // B
      cin.get(ch);
      cout << str << endl;
      cout << ch << endl;
}
```

设对该程序的响应为：

This is<CR>
a string#<CR>
c<CR>

则程序的输出为：

```
17
This is
a string
c
```

例 13.13 的行 A 调用函数 getline（）输入一个字符串，并指定其结束字符为 '#'，因此，输入中的回车符将作为字符串的有效字符被存入 str 中。由于输入串中字符 '#' 后边的字符仍留在输入缓冲区中（本例中只有一个回车符，读者在输入时可在回车符前多键入几个字符试试），因此程序在行 B 调用函数 ignore（）将缓冲区中多余的字符提取掉，以便往后的提取操作能够正常进行。

13.3.3 格式化输出

格式化输出是通过对类 ostream 的对象，利用重载了的运算符“<<”进行插入操作来实

现的。

最常用的格式化输出是对标准流 cout，cerr 和 clog 的插入。程序员也可以定义自己的输出流，但实用价值不大。

格式化输出缺省的格式为：十进制、域宽为 0、左对齐、空格填充。对于实数，缺省的精度为 6 位小数、浮点输出。但当一个实数的值超过某一范围——比如整数部分超过七位或有效数字在小数点后第四位以后——时，将自动转换为科学记数法输出。

例 13.14 对流 cout 插入。

```
//      EX13_14.CPP

#include      <iostream.h>
#include      <iomanip.h>

void   main()
{
       int     i = 123;
       float   f1 = 123.456, f2 = 12345678901234567.89;
       char    *str = "This is a string";

       cout << "12345678901234567890\n";
       cout << setw(10) << i << endl;
       cout << setw(3) << f1 << endl;
       cout << setprecision(2) << f2 << endl;
       cout << setw(20) << str << endl;
}
```

该程序的输出为：

```
12345678901234567890
       123
123.456
1.23e+16
    This is a string
```

标准输出流 cout 为缓冲流，但系统在每执行一条输出语句后会自动刷新该流。例如，对于语句：

```
cout << i;
```

尽管其中没有向流 cout 插入回车符，但当该语句执行完后，i 的值也会立即显示在屏幕上。对标准流 clog 则不同，系统不对该流自动刷新。这一点在对例 13.1 的说明中已作过介绍。

13.3.4 输出操作函数

与输入相对应，类 ostream 中也说明了一个用于输出字符的成员函数：

```
ostream&    ostream :: put(char);
```

该函数用于将参数所代表的字符插入到输出流中。其最大特点是可以将一个数值以其所对应的字符进行插入而不必进行强制类型转换。

例 13.15 使用 put（）函数。

```
//      EX13_15.CPP

#include      <iostream.h>

void   main()
{
       char    i = 65;

       while(i < 91)
              cout.put(i ++);
       cout.put('\n');
}
```

该程序的输出为

ABCDEFGHIJKLMNOPQRSTUVWXYZ

从附录可以看出，字符 A～Z 的 ASCII 码的十进制表示为 65～90。本例就是利用这一点来进行输出的。

在类 ostream 中，还说明了一个成员函数：

```
ostream&    ostream :: flush();
```

该成员函数为用户在需要时强制刷新一个输出流时提供了方便。

例 13.16 使用函数 flush（）。

```
//      EX13_16.CPP

#include      <iostream.h>

int     main(int argc, char *argv[])
{
       if(argc < 2)   {
              clog << "USAGE: EX13_1 someStrings<CR>" << endl;

              return 1;
       }
       clog << "Please enter an integer: ";                    // A
       clog.flush();                                           // B

       int   num;

       cin >> num;
       for(int i = 1; i < argc; i ++)
              cout << argv[i] << ' ';
       cout << endl;
       cout << "The number you entered is " << num << endl;
```

```
    return 0;
}
```

例 13.16 实际是例 13.1 的另一版本，为了在对输出进行重定向时行 A 的提示信息能在屏幕上显示，采用了对标准错误输出流 clog 的插入。而为了提示信息能及时地显示并保证用户输入与提示信息处于同一行，在行 A 没有插入回车符，在行 B 通过调用函数 flush（）达到这一目的。

对于标准流，一般情况下仅 clog 需要调用函数 flush（）来强制刷新流缓冲区。

13.3.5 重载提取和插入运算符

程序员可以重载运算符“>>”和“<<”，使提取和插入用户自定义的数据类型时可以像提取和插入 C++的基本数据类型一样的方便。

以提取和插入为目的重载这两个运算符时，必须将重载函数说明成类的友元。这是因为若将一个运算符重载为类的成员函数，则系统总是假定该运算符的左操作数为类的一个对象。这样，对于如下的语句：

```
cout << object;
```

由于 cout 不是对象 object 所属类的对象，而使得函数调用失败。

重载一个提取运算符应具有以下的一般形式：

friend istream& operator >> (istream&, class_name&);

即：函数必须返回对 istream 的引用，以便能用于链式输出中；函数的第一个参数也必须是对 istream 的引用，它将作为运算符的左操作数；第二个参数为对用户定义类类型的引用（也可以是类的对象），它将作为运算符的右操作数。

重载一个插入运算符应具有以下的一般形式：

friend ostream& operator >> (ostream&, class_name&);

即：函数必须返回对 ostream 的引用；函数的第一个参数也必须是对 ostream 的引用；第二个参数为对用户定义类类型的引用（也可以是类的对象）。

例 13.17 重载提取和插入运算符。

```
//      EX13_17.CPP

#include      <iostream.h>

class  C      {
              int     x, y;
       public:
              C(int a = 0, int b = 0) : x(a), y(b)     {}
              friend istream&       operator >> (istream&, C&);
              friend ostream&       operator << (ostream&, C&);
};

istream       &operator >> (istream &is, C &c)
```

```
{
        is >> c.x >> c.y;

        return is;
}

ostream         &operator << (ostream &os, C &c)
{
        os << c.x << ' ' << c.y;

        return os;
}

void    main()
{
        C       c1, c2(4, 5);

        cout << "C1: " << c1 << '\t' << "C2: " << c2 << endl;
        cout << "Please enter C1's x and y: ";
        cin >> c1;
        cout << "C1: " << c1;
}
```

经过以上的重载，当编译器遇到形如以下的表达式时：

cout << c1

就将它解释为

c1.operator << (cout, c1)

从而使得插入操作按程序员所定义的方式进行。

在定义重载的提取运算符“>>”时，只所以将该运算符的左操作数说明为对类 istream 的引用而不直接说明为对标准输入流 cin 的引用，是为了使该运算符也可以用于其他的输入流。在定义重载的插入运算符“<<”时，将左操作数说明为对类 ostream 的引用，也是出于这一目的。

习题

13-1 什么是流？什么是文件？

13-2 试说明缓冲的概念。

13-3 在习题 12-2 中重载了的插入运算符函数中加入适当的格式控制，使得输出整齐美观。

13-4 在习题 12-2 中重载了的提取运算符函数中增加错误检查机构，若用户输入有错，应能指出错误并给用户提供重新输入的机会。

第 14 章　文件流和内存流

本章介绍专门对磁盘文件进行 I / O 的文件流和对内存进行 I / O 的内存流。

14.1　文件流

在 C++语言的术语中，“文件”指的是一个具体的外部设备。而 C++的 I / O 流类体系中，定义有一种专门用于磁盘文件 I / O 的流类，习惯上称其为文件流。C++语言的文件流类体系在头文件 FSTREAM.H 中说明。

14.1.1　C++语言的文件流类体系

C++语言预定义的文件流类体系如图 14-1 所示。

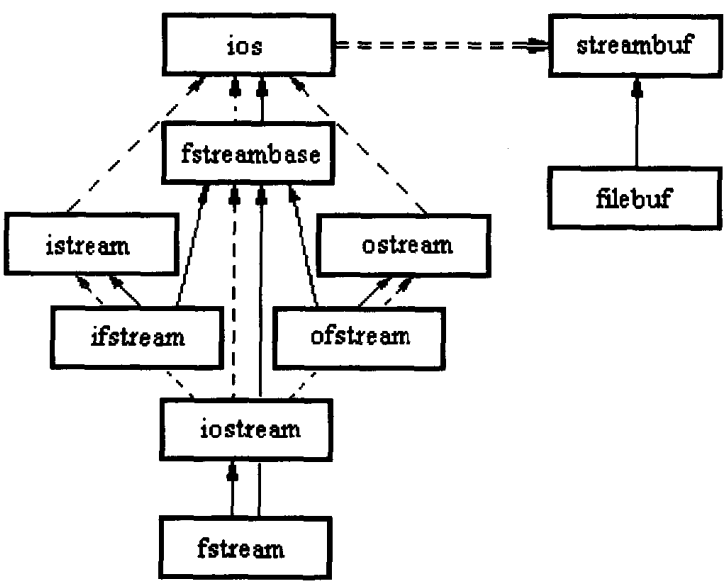

图 14-1　C++语言的文件流类体系

图 14-1 中：类 filebuf 用于管理文件的缓冲区，应用程序中一般不涉及该类；类 fstreambase 以类 ios 为其虚基类公有派生而来，它提供了文件 I / O 所需的所有成员函数；类 ifstream 从类 istream 和 fstreambase 多重公有派生，它用于从文件中输入数据；类 ofstream 从类 ostream 和 fstreambase 多重公有派生，它用于向文件中输出数据；类 fstream 从类 iostream 和 fstreambase 多重公有派生，它用于对文件进行双向传递数据的操作。后 3 个类中实际上没有定义新的成员。

14.1.2 文件的打开与关闭

C++语言没有提供预定义的文件流对象，所以，在访问一个文件流的对象时必须在程序中通过打开文件将该对象与一个具体的磁盘文件联系起来。而当不再访问该流时，应当通过关闭文件将对象与磁盘文件脱离联系。

14.1.2.1 打开文件

C++在文件流类中说明了4个打开文件的成员函数：

```
void  fstreambase :: open(char*, int, int = filebuf :: openprot);
void  ifstream :: open(char*, int = ios :: in, int);
void  ofstream :: open(char*, int = ios :: out, int);
void  fstreambase :: open(char*, int, int);
```

各函数中，第一个参数是一个字符串，该串给出可以带有全路经的磁盘文件名；第二个参数给出对所打开文件的访问模式；第三个参数给出所打开文件的保护方式，该参数与具体的操作系统有关，一般情况下只用它的缺省值 filebuf :: openprot。

在所有文件流的基类 ios 中，定义了一个文件打开模式的公有枚举成员：

```
enum open_mode {
      in          = 0x01,        // 以读（输入）为目的打开文件
      out         = 0x02,        // 以写（输出）为目的打开文件
      ate         = 0x04,        // 打开时将文件指针置于文件结尾处
      app         = 0x08,        // 以添加（输出）为目的打开文件
      trunc       = 0x10,        // 将文件的长度截为 0
      nocreate    = 0x20,        // 打开已存在文件
      noreplace   = 0x40,        // 打开不存在文件
      binary      = 0x80         // 打开二进制文件
};
```

这里，各模式是以位方式定义的，所以可以将几种模式或起来组合使用。

open_mode 各枚举常量中，in，out 和 binary 的意义已明确，这里对其他模式做进一步的解释。

ate 指出在打开文件后将文件读写指针置于文件结尾处。该模式必须与模式 in，out 或 noplace 同时使用，当与 out 同时使用时，它与 app 的意义相同。

app 实际上是以写为目的而打开文件的，它相当于 out | ate。当向以该模式打开的文件中直接输出时，所写入的信息将添加到文件的尾部。

以 trunc 模式打开文件时，若文件不存在，则系统将在磁盘上建立指定的文件。若文件已存在，则将文件的长度截为 0，这实际上就是将已存在文件的内容抹去。因此，使用该模式打开文件时应十分小心，防止不必要的损失。另外，当单独以 out 模式打开文件时，隐含有 trunc，也会将已存在文件中的信息删去。

nocreate 的意思是不建立新文件。因此，若欲打开的文件不存在，则打开文件的操作失

败。该模式也不单独使用，总是要说明是为读还是为写而打开文件的。另外，该模式不得与noreplace 同时使用。

noreplace 通常用来创建一个新文件。该模式也不单独使用，它总是以写而打开文件的。但是，若与 ate 或 app 同时使用，也可以打开一个已存在的文件。

另外，binary 也不单独使用，使用它时应指出为读或是为写而打开文件的。除非指定该模式，否则打开的均是文本流。

in 与 out 也可以同时使用，这时打开的文件既可以读又可以写。

上述 4 个 open（）函数从本质上讲只是一个函数，因为在后 3 个函数中实际上并没有定义具体的操作，只是采用不同的实参对第一个函数进行调用。然而，在应用程序中一般不使用类 fstreambase 的对象，因而在应用程序中经常调用的却是后 3 个函数。

类 ifstream 的方法 open（）缺省的文件打开模式为 in。因此，若 inflie 是一个类 ifstream 的对象，则语句

```
infile.open("myfile");
```

就将磁盘文件 MYFILE 打开用于输入。而语句

```
infile.open("myfile", ios :: binary);
```

则将磁盘文件 MYFILE 作为二进制文件打开用于输入。它相当于

```
infile.open("myfile", ios :: in | ios :: binary);
```

即，对于类 ifstream 的对象，打开文件时总是隐含有模式 in。

与此相似，调用类 ofstream 对象的 open（）方法时，总是隐含有模式 out。

由于类 fstream 的方法 open（）没有模式参数的缺省值，所以调用该类对象的 open（）方法时，必须在实参中明确给出文件的打开模式。

另外，在上述 4 个文件流类中都重载了一个带有缺省参数的构造函数：

```
fstreambase :: fstreambase(char*, int, int = filebuf :: openprot);
ifstream :: ifstream(char*, int = ios :: in, int);
ofstream :: ofstream(char*, int = ios :: out, int);
fstream :: fstream(char*, int, int);
```

调用这些构造函数时，也将同时打开文件。例如：

```
ifstream    infile("myfile");
```

不仅说明了一个类 ifstream 的对象 infile，同时该流还作为一个输入流与磁盘文件 MYFILE 联系起来。

无论是调用 open（）函数还是调用构造函数打开一个文件时，都可以通过调用基类 ios 中重载的取反运算符“!”或错误处理函数来检查文件打开的操作是否成功。一般来讲，在程序中必须做这样的检查，因为相对控制台 I / O 而言，磁盘文件 I / O 出错的可能性更大。

14.1.2.2 关闭文件

类 fstreambase 中说明了一个用于关闭文件的成员函数 clos（）：

```
void  fstreambase :: close();
```

调用该函数可以将一个流与一个具体的磁盘文件的联系断开。因为在 C++中文件流是缓冲流，因此关闭一个文件时，系统将刷新与该流相关的缓冲区。

当一个流的生存周期结束，系统在调用该流所属类的析构函数撤消该流时，首先关闭与该流相关的文件。因此，就是不调用函数 close（），在程序结束时所有打开的文件都将由系统自动关闭。这看来似乎该函数没有什么用处。事实上，函数 close（）的最大作用是可以将一个流与多个磁盘文件相联系。试分析下面的程序片断：

```
ifstream    infile("myfile");
// 对 myfile 的访问
infile.close();

infile.open("youfile");
// 对 youfile 的访问
```

这里，首先调用流 infile 的构造函数打开了文件 MYFILE。在对该文件的访问结束后，调用函数 close（）关闭 MYFILE。接着，程序又调用函数 open（）将流 infile 与文件 YOUFILE 联系起来，以便程序可以访问该文件。

应当说明的是：一个流任一时刻只能与一个磁盘文件发生联系，在与一个流相联系的文件没有关闭时，该流不得再打开另一个文件。

14.1.3 文件的访问

在 C++的文件流类中实际上没有定义什么文件操作函数，对文件的访问是通过调用各流类的基类——ios，istream 和 ostream——中所提供的方法来实现的。基类中说明的所有方法，包括提取运算符“>>”和插入运算符“<<”都可以用于对文件的访问。

例 14.1 使用文件流。

```
//      EX14_1.CPP

#include      <fstream.h>
#include      <stdlib.h>

void   main(int argc, char *argv[])
{
      if(argc != 3)   {
            cerr << "USAGE: EX14_1 SOURCE TARGET<CR>" << endl;

            exit(1);
      }

      ifstream       infile(argv[1]);

      if(!infile)     {
            cerr << "Con't open source file " << argv[1] << endl;

            exit(1);
```

```
    }

    ofstream      outfile(argv[2]);
    if(!infile)    {
          cerr << "Con't open target file " << argv[2] << endl;

          exit(1);
    }

    infile.unsetf(ios :: skipws);          // A

    char   ch;

    while(infile >> ch)
          outfile << ch;

    infile.close();
    outfile.close();
}
```

该程序是一个文本文件拷贝程序。设用户运行该程序时键入：

ex14_1 ex14_1.cpp ex14_1a.cpp<CR>

则磁盘上将产生一个名为 EX14_1A.CPP 的文本文件，其内容与本源程序完全相同。

注意，该程序中的行 A 是非常必要的。因为对于入流，缺省的格式控制为跳过前导的空白字符，若没有这一行则在新文件中所有的空白字符都将被删去，即新文件中的所有字符都连在一起。

在程序中，还可以通过说明类 fstream 的对象，然后根据需要调用函数 open（）将对象与某一文件联系起来。这种用法显得更为灵活。

例 14.2 例 14.2 的另一版本。

```
//      EX14_2.CPP

#include      <fstream.h>
#include      <stdlib.h>

void   main(int argc, char *argv[])
{
      if(argc != 3)   {
            cerr << "USAGE: EX14_2 SOURCE TARGET<CR>" << endl;

            exit(1);
      }

      fstream        infile, outfile;

      infile.open(argv[1], ios :: in);
      outfile.open(argv[2], ios :: out);
      if(!infile | !outfile)    {
            cerr << "Files open failure!" << endl;
```

```
        exit(1);
    }

    char    ch;

    while(infile.get(ch))
        outfile.put(ch);

    infile.close();
    outfile.close();
}
```

本例中，由于使用了函数 get（）和 put（），因此可以不必担心文件中的空白字符被删除掉。

14.2 几个主要用于文件的函数

在文件流类的几个基类中还说明了一些主要用于文件操作的成员函数，这些函数一般不用于控制台 I / O。

14.2.1 文件的读写函数

在类 istream 中重载了两个 read（）成员函数：

```
istream&    istream :: read(signed char*, int);
istream&    istream :: read(unsigned char*, int);
```

它们第一个参数为一个字符型数组，用来存放从流中读取的数据；第二个参数指出欲读取的字节数。

在类 ostream 中重载了两个 write（）成员函数：

```
ostream&    ostream :: write(signed char*, int);
ostream&    ostream :: write(unsigned char*, int);
```

其第一个参数中存放着欲写入到流中的数据，第二个参数给出欲写的字节数。

这两组函数主要用于对二进制文件流的 I / O。

由于在类 ios 中所定义的、表示文件结束的宏 EOF 的值为 -1，而在二进制文件中 -1 可能是一个有效的数据。为此，类 ios 中还说明了一个专门测试文件是否结束的成员函数：

```
int     ios :: eof();
```

该函数返回非零值表示文件已经结束。它不仅可以用来测试二进制文件，也可以用来测试文本文件是否结束。

例 14.3 例 14.2 的二进制版本。

```
//      EX14_3.CPP

#include      <fstream.h>
#include      <stdlib.h>
```

```
void    main(int argc, char *argv[])
{
        if(argc != 3)   {
                cerr << "USAGE: EX14_3 SOURCE TARGET<CR>" << endl;

                exit(1);
        }

        fstream         infile, outfile;

        infile.open(argv[1], ios :: in | ios :: binary);
        outfile.open(argv[2], ios :: out | ios :: binary);
        if(!infile | !outfile)     {
                cerr << "Files open failure!" << endl;

                exit(1);
        }

        char ch[1024];

        do      {
                infile.read(ch, 1024);
                if(infile.eof()) {
                        int    n = infile.gcount();

                        outfile.write(ch, n);

                        break;
                }
                else
                        outfile.write(ch, 1024);
        } while(1);
}
```

该程序利用方法 read（）和 write（）每次读写 1 024 个字节，使程序的效率大大提高。读者可以分别用程序 EX14_2 和 EX14_3 来拷贝同一个文件，比较一下它们的执行速度。

注意程序中 do-while 循环中的 if 分支：这里首先检查输入文件是否结束，若是，则调用函数 gcount（）获得本次提取的实际字节数，然后按实际读取的字节数将数组 ch 中的内容写入到输出文件中去。所以这样做是因为函数 read（）在提取出错时（提取到文件结束标志将被视为一个错误）将结束执行。因此，若遇到文件结束，读取到数组 ch 中的数据可能不足 1 024 个字节。该分支的执行还将导致 do-while 循环的终止，从而使得整个程序的运行结束。

需要说明的是：程序 EX14_3 不仅可以用来拷贝二进制文件，用它拷贝文本文件时也不会有任何问题。反之，有些专门为文本文件设计的程序，用于二进制文件时将会出现问题。

在类 istream 中还说明两个成员函数：

```
int     istream :: peek();
istream&     istream :: putback(char);
```

函数 peek（）返回输入流中下一个字节，但并不提取它。比如：

```
//...
ch1 = infile.peek();
infile.get(ch2);
if(ch1 == ch2)
        //...
```

则 if 分支得以执行。这是由于 ch1 从输入流中获得的值时，该值并没有从流中提取，当调用函数 get（）从流中提取数据时，ch2 得到的是同一值之故。

函数 putback（）将其参数所表示的 1 个字节返还到输入流中。这就是说，将该函数的实参放到输入流中的当前位置之前，并将文件指针退回 1 个字节指向该字节。这样，下一次提取的第一个字节就是这个被返还的字节。

例 14.4 从输入文件中读取 C++标识符的函数。

```
//      EX14_4.CPP

#include      <fstream.h>
#include      <ctype.h>

void   GetIdent(ifstream infile, char *s)
{
       char   ch;

       infile >> ch;                          // 跳过前导空白
       if(isalpha(ch) || ch == '_')           // 若第一个有效字符是字母或下划线
              do     {                        // 读到的字符是字母、下划线或数字则循环
                     *s ++ = ch;
                     infile.get(ch);
              } while(isalnum(ch) || ch == '_');

       *s = 0;                                // 加上字符串结束标志
       infile.putback(ch);                    // 将多读的一个字符放回流中
}
```

该函数可以用来在一个源程序文件中读取 C++语言的标识符。该函数从流 infile 中读取一个个的字符，若第一个有效字符不是字母或下划线，则说明流中当前的字符不属于一个标识符。因此，将该字符放回到流中以便其它的分析 C++词法符号的函数来读取它，同时将串 s 置为空串。若当前字符属于一个标识符，则利用 do-while 循环将该标识符逐一地读入字符串 s 中，直至读到一个不属于标识符的字符为止。这时，为字符串加上结束标志并将多读的一个字符放回到输入流中。

在本例中，用到了两个 C++语言的库函数，它们的原型在头文件 ctype.h 中说明为：

```
int    isalpha(int c);
int    isalnum(int c);
```

第一个函数，若其参数 c 的值是一个字母字符的 ASCII 码则返回非零值；第二个函数，若其参数 c 的值是一个字母字符或是一个数字字符的 ASCII 码则返回非零值。

14.2.2 文件随机访问的函数

由于磁盘文件是允许随机访问的，所以在文件流类的基类中还说明了几个用于对文件进行随机访问的成员函数：

```
istream&    istream :: seekg(streampos);
istream&    istream :: seekg(streamoff, seek_dir);
streampos   istream :: tellg();
ostream&    ostream :: seekp(streampos);
ostream&    ostream :: seekp(streamoff, seek_dir);
streampos   istream :: tellp();
```

其中：参数的数据类型 streampos 和 streamoff 在类 ios 中定义为

```
typedef     long  streampos;
typedef     long  streamoff;
```

即，它们就是 long；而 seek_dir 则是类 ios 中说明的一个公有枚举成员：

```
enum seel_dir     {
      beg   = 0,          // 文件开始处
      cur   = 1,          // 当前位置
      end   = 2           // 文件结束处
};
```

各 seek（）函数是用来将文件指针移到流中指定位置的。其中 g 是 get 的缩写，表示移动输入流的文件指针（读指针）；p 是 put 的缩写，表示移动输出流的文件指针（写指针）。只带有一个参数的 seek（）函数可以用来将文件指针移动到参数所给出的字节数处。而带有两个参数的 seek（）函数则可以用来从指定位置向文件尾部或首部（当第一个参数取小于零的值时）移动第一个参数所给出的字节数。这里只所以将代表移动字节数的参数说明为 long 型，是为了可以在文件长度大于 64KB 的文件中随机移动文件指针。函数 tellg（）和 tellp（）分别返回输入流中和输出流中文件指针的当前位置。

应当指出的是：任何将文件指针移到文件起始处以前或文件结束标志以后的操作都将导致往后的 I / O 操作的失败。保证文件指针不被移出文件是程序员的责任。

下面举例说明这几个函数的用法。

```
infile.seekg(100);                    // 将文件指针移动到文件中第 100 个字节处
outfile.seekp(sizeof(Person), ios :: beg);
            // 从文件起始处向尾部移动一个类 Person 对象所占的字节数
inflie.seekg(-20, ios :: cur);        // 从当前位置向头部移动 20 个字节
outfile.seekp(-10, ios :: end);       // 从文件尾部向头部移动 10 个字节
long  pos = infile.tellg();           // 取得当前位置
```

14.3 文本文件和二进制文件

磁盘文件不同于控制台文件和打印机文件，它不仅可以以文本形式存储信息，还可以以二进制形式存储信息。两种形式各有其优缺点，实用中应根据信息自身的特点来确定其存储形式。一般讲，对于纯文本信息（比如字符串），以文本形式存储较佳；而对于数值信息，则最好存为二进制形式。

14.3.1 文本文件

将数据以文本形式存盘，其优点是文件具有较高的兼容性。文本文件可以利用任何一个文字处理程序进行阅读、编辑、修改和打印，在 DOS 环境下，还可以利用 TYPE 命令将其内容输出到显示器或打印机。

然而，文本文件也存在一些缺点：对于纯数值文件，存储时必须在数据之间人为地添加分隔符，否则将会导致数据无法正确读出；另外，文本文件通常要比对应的二进制文件所占的磁盘空间大，且不便于对数据的随机访问。

例 14.5 将数值数据存为文本文件。

```
//      EX14_5.H

int     *GenerateDatas(int);
void    ReleaseMemory(int*);
void    SortData(int*, int);


//      EX14_5A.CPP

#include      <fstream.h>
#include      <stdlib.h>
#include      <time.h>
#include      "ex14_5.h"

int     *GenerateDatas(int n)
{
        int     *ip;

        ip = new int[n];
        if(!ip)
                return 0;

        randomize();
        for(int i = 0; i < n; i ++)
                ip[i] = rand() % 1000;

        return ip;
}
```

```
void    ReleaseMemory(int *ip)
{
        delete ip;
}

void    SortData(int *ip, int n)
{
        int     i, j, flag, temp;

        for(i = 0; i < n - 1; i ++)       {
                flag = n - 1;
                for(j = n - 1; j > i; j --)
                        if(ip[j] < ip[j - 1])     {
                                temp = ip[j];
                                ip[j] = ip[j - 1];
                                ip[j - 1] = temp;

                                flag = j;
                        }

                if(flag == n - 1)
                        break;
        }
}


//      EX14_5.CPP

#include      <iostream.h>
#include      <fstream.h>
#include      <stdlib.h>
#include      <conio.h>
#include      "ex14_5.h"

void    main()
{
        int     SaveData(int*, int, char*);

        int     *iArray;

        iArray = GenerateDatas(10);
        if(!iArray)     {
                cout << "Memory allocation error, press any key to exit.\n";
                getch();
                exit(1);
        }

        SortData(iArray, 10);
        SaveData(iArray, 10, "DATAS.DAT");

        ReleaseMemory(iArray);
```

```
}

int     SaveData(int *ip, int n, char *fname)
{
        ofstream        of(fname);

        if(!of) {
                cout << "File creatting error!" << endl;

                return 0;
        }

        for(int i = 0; i < n; i ++)
                of << ip[i] << ' ';

        of.close();

        return 1;
}
```

该程序产生 10 个 1 000 以内的随机整数，并将它们以文本形式存入磁盘文件 DATAS.DAT 中。程序运行成功后，磁盘的当前目录下将产生一个包含以下内容的文本文件：

```
16 567 710 719 756 795 886 895 937 975
```

注意程序 EX14_5.CPP 中函数 SaveData（）的输出语句：

```
of << ip[i] << ' ';
```

其作用是在向文件中插入一个数据后，立即再插入一个空格字符作为分隔符。

例 14.5 中用到的函数 randomize（）是一个 C++语言的库函数，该函数的原型在头文件 STDLIB.H 中被说明为

```
void  randomize(void);
```

这是一个即无参数也无返回值的函数。该函数的作用是利用系统时钟对 C++软件包的随机数生成器进行初始化，以提高 rand（）函数所产生的随机数的随机度。

文件 DATAS.DAT 中的数据可以利用例 14.6 中的程序读出。

例 14.6 读取文本文件中的数据。

```
//      EX14_6.CPP

#include      <iostream.h>
#include      <fstream.h>

void    main()
{
        int     LoadData(int*, char*);
        void    ShowData(int*, int);

        int     iArray[10], i;
```

```
    i = LoadData(iArray, "DATAS.DAT");
    ShowData(iArray, i);
}

int     LoadData(int *ip, char *fname)
{
    ifstream        ifs(fname);
    int     i = 0;

    if(!ifs) {
        cout << "File openning error!" << endl;

        return 0;
    }

    while(1)        {
        ifs >> *ip;
        if(ifs.eof())
            break;
        ip ++;
        i ++;
    }

    return i;
}

void    ShowData(int *ip, int n)
{
    for(int i = 0; i < n; i ++)
        cout << ip[i] <<' ';
    cout << endl;
}
```

该程序的输出与其内容完全一样。

文件的读取操作是通过一个 while 循环来完成的：

```
while(1)    {
    ifs >> *ip;             // 从文件中读取一个整数存入 ip 所指的内存中
    if(ifs.eof())           // 若文件已经结束则退出循环
        break;
    ip ++;                  // ip 指向下一个元素
    i ++;                   // 读取的数据个数加 1
}
```

从该循环的循环体中可以看出，在文本文件中读取一个数据（本例是一个整数）是非常方便的，只需在文件的当前位置处将数据提取到一个变量中即可。而实际读取的数值则与从控制台提取数据的操作完全相同。

应当说明的是：本例所以能够利用如此简单的操作从磁盘文件中读取数据，完全是因为

数据在存盘时利用了空白字符作为数据的分隔符。如果使用其它字符作为分隔符，比如逗号，则必须在读取一个数据之后再空读一个字符以将文件的读指针移到逗号的后边。

从数据在文件中的存储形式可以看出，数据在文件中是不等长的，这就妨碍了对文件的随机访问。

14.3.2 二进制文件

二进制形式存储的信息其缺点是无可读性，且兼容性较差。而其优点是节省磁盘空间，读写操作简单快捷。

例 14.7 将数值数据存为二进制文件。

```
//      EX14_7.CPP

#include      <iostream.h>
#include      <fstream.h>
#include      <stdlib.h>
#include      <conio.h>
#include      "ex14.h"

void    main()
{
        int     SaveData(int*, int, char*);

        int     *iArray;

        iArray = GenerateDatas(10);
        if(!iArray)     {
                cout << "Memory allocation error, press any key to exit.\n";
                getch();
                exit(1);
        }

        SortData(iArray, 10);
        SaveData(iArray, 10, "DATAS1.DAT");

        ReleaseMemory(iArray);
}

int     SaveData(int *ip, int n, char *fname)
{
        ofstream        of;

        of.open(fname, ios :: out | ios :: binary);
        if(!of) {
                cout << "File creatting error!" << endl;

                return 0;
        }
```

```
    of.write((char*)ip, n * sizeof(int));          // A

    of.close();

    return 1;
}
```

例 14.7 是例 14.5 的二进制版本。从该例可以看出：将数据以二进制形式存盘，其操作要简单得多。就本例而言，仅需一条语句（程序中的行 A），且无需添加分隔字符。

例 14.8 读取二进制文件中的数据。

```
//      EX14_8.CPP

#include      <iostream.h>
#include      <fstream.h>

void    main()
{
    int     LoadData(int*, char*);
    void    ShowData(int*, int);

    int     iArray[10], i;

    i = LoadData(iArray, "DATAS1.DAT");
    ShowData(iArray, i);
}

int     LoadData(int *ip, char *fname)
{
    ifstream        ifs;
    int     i = 0;

    ifs.open(fname, ios :: in | ios :: binary);
    if(!ifs) {
        cout << "File openning error!" << endl;

        return 0;
    }

    while(1)        {
        ifs.read((char*)ip, sizeof(int));
        if(ifs.eof())
            break;
        ip ++;
        i ++;
    }

    return i;
}

void    ShowData(int *ip, int n)
{
    for(int i = 0; i < n; i ++)
```

```
            cout << ip[i] << ' ';
        cout << endl;
}
```

例 14.8 是例 14.6 的二进制版本，它从一个存放着二进制整数的磁盘文件 DATA1.DAT 中将数据逐个读到一个整型数组中。若事先知道数据的个数，还可以一次将所有的数据全部读入。

由于二进制文件中的信息是数据的二进制映像，因此其长度是固定的。比如，一个整型数据占两个字节、一个实型数据占 4 个字节，等等。这就使得对文件的随机访问成为可能。

例 14.9 磁盘文件的随机访问。

```
//      EX14_9.CPP

#include      <iostream.h>
#include      <fstream.h>
#include      <conio.h>

void    main()
{
        long    SearchData(int, char*);

        int     data;
        long    loc;

        cout << "Plase enter data which you want to search: ";
        cin >> data;
        loc = SearchData(data, "DATAS1.DAT");
        if(loc == -1)
                cout << "Not fond!\n";
        else
               cout << loc << endl;
}

long    SearchData(int d, char *fname)
{
        fstream         ifs;
        streampos       low = 0, hi, mid;
        int     dat;

        ifs.open(fname, ios :: in);
        if(!ifs) {
                cout << "File openning error!" << endl;
                getch();

                return -1;
        }

        ifs.seekg(0, ios :: end);
        hi = ifs.tellg();
        hi /= sizeof(int);
        while(low <= hi)      {
```

```
            mid = (hi + low) / 2;
            ifs.seekg(mid * sizeof(int));
            ifs.read((char*)&dat, sizeof(int));
            if(dat == d)
                return mid;
            if(dat > d)
                hi = mid - 1;
            else
                low = mid + 1;
    }

    return -1;
}
```

例 14.9 要求用户输入一个整数，然后利用二分搜索在文件 DATA1.DAT 中查找与该数相等的数据。若文件中存在该数据，则输出它在文件中的位置；否则输出一个信息“Not fond!”。

二分搜索的基本思想是：在一个有序的数据集中，逐步将搜索的范围减半，直到查找成功或搜索范围为 0。其具体操作为：首先将数据集中位于中央的那个数据与欲查找的数据进行比较，若两数相等，则查找成功；否则：若当前数据小于欲查找的数据，则在数据集的前半部分（设数据为升序排列的）进行二分搜索；若当前数据大于欲查找的数据，则在数据集的后半部分进行二分搜索。

14.4 内存流

C++语言的 I / O 流类不仅可以将外部设备作为 I / O 对象，还可以将数组变量作为 I / O 对象，即对一块内存进行输入输出。这种对内存进行 I / O 的流常被称作内存流、字符串流、数组流或 RAM 流。C++语言的内存流类体系在头文件 STRSTREAM.H 中说明。

14.4.1 C++的内存流类体系

C++语言预定义的内存流类体系如图 14-2 所示。

从图 14-2 可以看出，内存流类体系与文件流体系的派生过程完全一样，这里不再赘述。

应当文说明的是：各内存流类中除了构造函数外几乎没有再说明什么新的成员。在应用程序中，内存流使用的全都是从它们的基类 ios，istream 和 ostream 中继承来的方法。

14.4.2 使用内存流

在类 istrstream，ostrstream 和 strstream 中，除了说明有缺省的构造函数以外，还分别重载了一个带参的构造函数：

```
istrstream :: istrstream(char*, int);
ostrstream :: ostrstream(char*, int, int = ios :: out);
strstream :: strstream(char*, int, int);
```

其中：第一个参数是一个用来创建内存流的字符数组，即把一个内存流与该字符数组联系起

来；第二个参数为一整数，它指出用字符数组中的多少个字节来创建指定的内存流；第三个参数给出内存流的打开模式。对于输出流类 ostrstream，第三个参数的缺省值为 ios :: out。对于输入流则没有第三个参数，即系统认为该字符数组中已经存放有数据，可以直接从中提取。

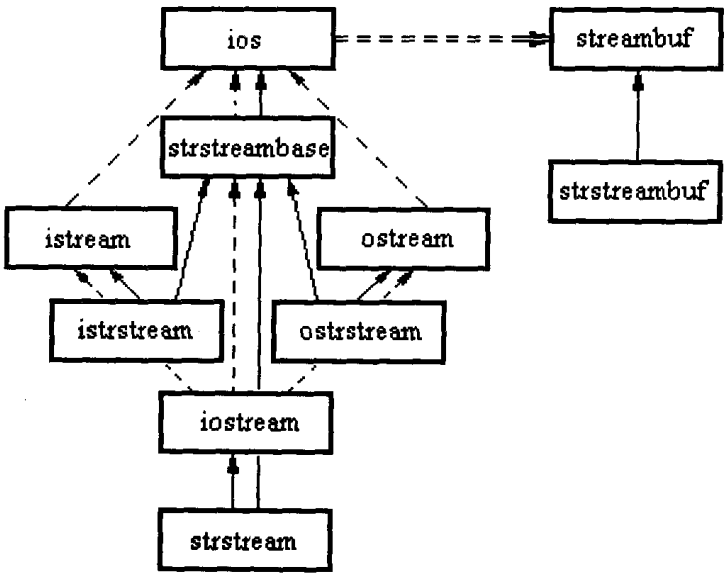

图 14-2　C++语言的内存流类体系

实际上，istrstream 还说明了一个只有一个参数的构造函数：

istrstream :: strstream(char*);

使用该构造函数创建流时，系统将把数组与流联系起来，其长度由数组中实际存在的字符个数确定（包括字符串结束标志）。

一般讲，当用一个字符型数组创建一个输入内存流时，应当保证数组中已存在一个有效的字符串。

例 14.10　使用内存流。

```
//      EX14_10.CPP

#include      <strstrea.h>
#include      <iomanip.h>

void    main()
{
        char    str[81];
        char    ch = 'a';
        ostrstream      outstr(str, sizeof(str));

        for(; ch <= 'z'; ch ++)
                outstr << ch;
        outstr.setf(ios :: hex);
        for(int i = 1; i < 20; i += 2)
                outstr << setw(4) << i;
        outstr << ends;                         // A
```

```
    cout << str << endl;

    istrstream      instr(str);
    int   j;

    instr.setf(ios :: hex);
    instr.seekg(27);
    for(int i = 0; i < 10; i ++)    {
        instr >> j;
        cout << j << '\t';
    }
    cout << endl;
}
```

该程序的输出为：

```
abcdefghijklmnopqrstuvwxyz    1    3    5    7    9    b    d    f   11   13
1    3    5    7    9    11    13    15    17    19
```

程序中，行 A 向内存流 outstr 中利用操纵符 ends 插入一个字符串结束标志。该操作是十分必要的，否则随后的将 str 向控制台输出将会出现混乱。

14.5 两个专门用于内存流的函数

在类 ostrstream 中说明了两个公有成员函数：

```
int     ostrstream :: pcount();
char* ostrstream :: str();
```

函数 pcount（）返回输出到内存流中字符的个数，该数中包括空白字符和字符串的结束标志（如果有的话）。例如，若在例 14.5 的行 A 后边加一条语句：

```
i = outstr.pcount();
```

则 i 的值为 67。

函数 str（）返回与一个内存流相联系的字符型数组的首地址。一旦调用一个流的 str（）函数，则与该流相联系的字符数组就与流脱离联系。该函数主要用于返回一个在创建流时动态建立的数组。

当调用类 ostrstream 缺省构造函数创建一个内存流时，该构造函数就建立一个动态数组。所创建数组的大小不定，因而特别适合创建预先不知其大小的内存流。由于与该流相联系的数组是系统自动建立起来的，它没有名子，因而也就无法获取其中的数据。利用函数 str（）将该数组与内存流脱离联系，同时返回数组的首地址，这样，程序就可以像访问普通数组那样访问该数组中的数据了。

例 14.11 使用动态数组。

```
//      EX14_11.CPP

#include      <strstrea.h>
```

```
#include    <fstream.h>

int    main(int argc, char *argv[])
{
    if(argc != 2)   {
        cerr << "USAGE: EX14_6 file_name<CR>" << endl;

        return 1;
    }

    ifstream    infile(argv[1]);

    if(!infile)    {
        cerr << "Con't open file " << argv[1] << endl;

        return 2;
    }

    ostrstream    outstr;                    // A
    char    ch[1024];

    do    {
        infile.read(ch, 1024);
        if(infile.eof()) {
            int   n = infile.gcount();

            outstr.write(ch, n);

            break;
        }
        else
        outstr.write(ch, 1024);
    } while(1);

    outstr << ends;

    char    *s = outstr.str();                 // B

    cout << s << endl;

    delete s;                                  // C

    return 0;
}
```

设运行该程序时键入：

ex14_11 ex14_10.cpp<CR>

则源程序 EX14_10.CPP 将显示在屏幕上。

程序中行 A 调用类 ostrstream 的缺省构造函数创建了一个内存流 outstr，这时我们并不

知道与该流相联系的数组是什么。程序中的 do-while 循环与例 14.3 相似，该循环的功能是从磁盘文件中读取数据。与例 14.3 不同的是，这里将读得的数据输出到一个内存流中而不是文件流中。当数据输入操作完毕后，程序中行 B 就调用流 outstr 的成员函数 str（），将与该流相关的字符型数组的首地址返回给字符型指针 s，然后将该数组的内容插入到标准输出流中。

注意程序中的行 C，该行将动态数组所占据的内存单元返还给操作系统。一般情况下，程序员有责任将由调用函数 str（）而获得的动态数组 delete 掉。

习题

14-1 为什么在使用文件前先得打开文件，而在文件使用后又必须关闭文件？

14-2 简述文本文件与二进制文件的区别。

14-3 向文本文件中写入数值数据时应注意些什么？

14-4 说明一个习题 9-6 中所定义之大小可变的整型数组的对象，并随意为该对象的元素赋一些值。设计一个函数：

```
int    SaveData(char *FileSpec, Array &arr);
```

将该数组（由参数 arr 指定）中有效元素存入参数 FileSpce 所指定的文本文件中，函数的返回值为实际写入文件中数据的个数，若文件操作失败则返回 0。

提示：由于数组中实际元素的个数与数组的大小不一定相等，且为了方便以后读取数据的操作，应将实际元素的个数作为第一个数据存盘。

14-5 将习题 14-4 改写成二进制文件形式。

14-6 编写一个函数：

```
int    LoadData(char *FileSpec, Array &arr, int bin);
```

根据参数 bin 的值，分别将习题 14-4（bin = 0）和习题 14-5（bin = 1）所存盘的数据读入到对象 arr 中。函数返回值指出读盘操作成功与否。

14-7 为类 Array 重载插入和提取运算符，使其自身具有存盘和读盘的能力。

14-8 将习题 14-4 中存盘的数据读入一个内存流中，并将该内存流中的内容显示在屏幕上。

14-9 说明一个 10 个元素的 Person 类的数组，并为其所有元素赋值。将该数组按年龄排序后存盘。根据用户输入的年龄范围，将所有符合条件之人的数据从文件中读出并显示出来。

第 15 章　模板

模板（Template）是 C++语言的代码重用性和多态性的一个集中表现。模板提供这样一个转换机制：由程序员定义一种操作或一个类，而该操作或类却可以适应几乎所有的数据类型。在一定意义上，模板类似宏定义或函数重载，但是它更加灵活，书写更为简洁，适应性更强。

模板可分为函数模板和类模板。前者为程序员编写通用的函数提供了一种手段；而后者则为程序员设计通用类奠定了基础。

应当说明的是：ANSI C++标准虽然规定了模板的各种技术规范，但却没有指定它的实现方法，这就使得不同的具体实现难以保证兼容。所以，包含有模板的源程序一般不具备移植性。

15.1　函数模板

函数模板可以用来定义通用的函数。它的作用就像函数的重载，但其编码却要比函数重载简便得多。

15.1.1　定义模板函数

利用函数模板定义的函数叫做模板函数。定义一个模板函数具有以下的一般形式：

```
template<class T>
type   Function_Name(arg_list)
{
      // Function_Body;
}
```

其中：关键字 template 指出下面将要说明的是一个模板；而<class T>则是该模板所要求的参数。这里的关键字 class 与类无关，而是与参数类型名 T 一起说明“这是一个用户定义的数据类型”，它和一对尖括号是模板语法的组成部分。参数类型名 T 可以是任何一个合法的标识符。

例 15.1　定义模板函数。

```
//      EX15_1.H

template<class T>
T       Max(T a, T b)
{
```

```
    return (a > b) ? a : b;
}

template<class T>
T    Min(T a, T b)
{
    return (a < b) ? a : b;
}
```

在头文件 EX15_1.H 中，定义了两个模板函数——Max（）和 Min（），它们分别返回其参数 a 与 b 中的大者和小者，而这两个参数的类型则取决于调用时实参的数据类型。

应当说明的是，在模板函数的参数表中，至少有一个参数的类型为模板的参数类型 T。另外，函数的返回值的类型也可以是 T，如上例所示。

定义模板函数时应注意：

1．模板中可以带有多个参数类型。比如：

```
template<class T1, class T2, class T3>
int    f(T1 arg1, T2 arg2, T3 arg3)
{
    //...;
}
```

2．模板的参数类型也可以是一个已存在的数据类型。比如：

```
template<class T, int n)
T    f(T arg)
{
    int    i = n;
    //...;
}
```

3．函数可以带有模板参数表中未给出的、已存在的数据类型的参数。比如：

```
template<class T>
T    f(T arg, int n)
{
    //...;
}
```

例 15.2 排序模板函数。

```
//    EX15_2.H

#include    <iostream.h>

template<class T>
int    BubbleSort(T *Arr, int n)
{
    int    i, j, flag;
```

```
    for(i = 0; i < n - 1; i ++)       {
        flag = n - 1;
        for(j = n - 1; j > i; j --)
            if(Arr[j] < Arr[j - 1]) {
                T       temp = Arr[j];

                Arr[j] = Arr[j - 1];
                Arr[j - 1] = temp;

                flag = j;
            }

        if(flag == n - 1)
            break;
    }

    return i + 1;
}

template<class T>
void    ShowData(T *Arr, int n)
{
    for(int i = 0; i < n; i ++)
        cout << Arr[i] << endl;
}
```

在头文件 EX15_2.H 中，定义了一个通用的冒泡排序模板函数 BubbleSort（），该函数可以对几乎所有数据类型的数组进行排序。函数的第一个参数 Arr 即为欲排序的数组，而第二个参数 n 则给出了数组的大小。模板函数 ShowData（）可以用来将一个任意类型数组的各元素输出到屏幕上。

15.1.2 使用模板函数

有了 EX15_1.H 中的函数定义，我们就可以很方便地得到两个类型相同的数据之最大值和最小值，而不管它们的数据类型为何。

例 15.3 使用 Max（）和 Min（）模板函数。

```
//      EX15_3.CPP

#include    <iostream.h>
#include    "ex15_1.h"

void    main()
{
    int     i1 = 3, i2 = 5;
    float   f1 = 3.141159, f2 = 2.71828;

    cout << Max(i1, i2) << endl;
    cout << Min(f1, f2) << endl;
    cout << Max('a', 'b') << endl;
```

```
}
```

该程序的输出为：

```
5
2.81929
b
```

从例 15.3 可以看出：当实参的数据类型为整型时，编译器就将模板中的 class T 解释成 int；当实参的数据类型为实型时，编译器就将 class T 解释成 float；而当实参的数据类型为字符型时，编译器则将 class T 解释成 char。这是 C++语言多态性的又一体现。注意：模板所体现的的多态性为编译时的多态性。

当然，该例的功能可以利用函数重载来实现。但使用函数重载需要编写多个函数，而利用模板则只需要一个函数即可。事实上，这两个模板函数还可以对长整型、短整型、双精度浮点型、甚至某些类类型——比如，字符串类——进行操作。C++语言的代码重用性在这里得到了充分的表现。

有了 EX15_2.H 中的函数定义，程序中就可以对任意数据类型的数组进行排序。

例 15.4 使用 BubbleSort（）模板函数。

```
//      EX15_4.CPP

#include      <iostream.h>
#include      <stdlib.h>
#include      "ex12_15.h"
#include      "ex15_2.h"

void    main()
{
        int     iArr[10], i;
        char    *strs[] = {"China", "America", "France", "Germany", "Jap an"};
        String sArr[5];

        for(i = 0; i < 10; i ++)
                iArr[i] = rand() % 1000;

        for(i = 0; i < 5; i ++)                         // A
                sArr[i] = strs[i];

        BubbleSort(iArr, 10);
        ShowData(iArr, 10);

        BubbleSort(sArr, 5);
        ShowData(sArr, 5);
}
```

该例的输出为：

```
90
117
126
```

```
130
346
415
595
656
948
982
America
China
France
Germany
Japan
```

注意：由于本例用到了第 12 章中定义的字符串类的对象，所以编译该程序时必须连接 EX12_15.OBJ，或将 EX12_15.CPP 与 EX15_4.CPP 放在同一个工程文件中。

注意程序中的循环 A，它的作用是用字符串数组 strs 为类 String 的对象数组 sArr 进行赋值。为什么不直接对 strs 数组进行排序呢？这是由于在模板函数 BubbleSort（）中，为了判断数组中相邻两个元素的位置是否正确，需要利用关系运算符“<”来比较它们的大小。而数组 strs 中的元素均为普通字符串，比较它们的大小需调用库函数 strcmp（）。这与模板函数中的操作不一致。而在类 String 中重载了关系运算符“<”，使得模板函数可以正确地对它的对象进行排序。因此，调用模板函数时，必须保证实参满足函数施加在它上面的操作。以冒泡排序函数模板为例，它的实参必须是可以利用关系运算符“<”正确地进行比较的。

15.5.3 特例函数

头文件 EX15_1.H 中所定义的两个函数存在一个缺陷，那就是不能正确地处理字符串（注意，不是指第 12 章中所说明的字符串类的对象。实际上，冒泡排序模板函数也存在这一缺陷，如上节所述）。比如，对于如下的说明：

```
char   *str1 = "The C", *str2 = "The C++";
```

则语句：

```
cout << Max(str1, str2) << endl;
```

的输出可能是：

```
The C
```

这是由于当用两个指针作为实参调用模板函数 Max（）时，函数比较的是两个指针的大小，而不是它们所指的内容的大小。这一点可以从函数的定义看出。

为解决类似的问题，C++语言提供了一种称为特例（Specializtion）的机制。所谓特例函数，实际上就是一个一般的函数。C++编译器在处理一个函数调用时，首先检查是否存在一个一般的函数与该函数调用相匹配，若存在这样的函数，则调用该函数；否则调用模板函数。也就是说，一般函数支配模板函数。

例 15.5 特例函数。

```
//      EX15_5.CPP

#include      <iostream.h>
#include      <string.h>
#include      "ex15_1.h"

char    *Max(char *s1, char *s2)
{
        if(strcmp(s1, s2) >= 0)
                return s1;
        else
                return s2;
}

void    main()
{
        char    *str1 = "The C", *str2 = "The C++";

        cout << Max(10, 5) << endl;
        cout << Max(str1, str2) << endl;
}
```

例 15.5 中所定义的 Max（char*, char*）函数就是一个特例函数，它明确地定义了当实参为字符型指针时，Max（）函数的具体操作。程序中的

```
cout << Max(str1, str2) << endl;
```

实际上调用的就是该函数，而不是模板函数 Max（T, T）。

也可以为冒泡排序模板函数编写特例函数，以使该函数能够对普通字符串数组进行排序。然而，利用 C++语言编程，还是定义一个完善的字符串类，所有涉及到字符串的数据和操作均使用该类的对象。这才是较佳的选择。

15.2 类模板

类是对问题空间的抽象，而类模板则是对类的抽象，即更高层次上的抽象。与函数模板相似，程序中可以通过高度抽象首先定义一个模板类，然后通过使用不同的实参生成不同类的对象。

15.2.1 定义模板类

定义一个模板类具有以下的一般形式：

```
template<class T>
class  Class_Name      {
       members;
};
```

其中：Class_Name 为所定义的模板类的类名。

模板类的具体内容与普通类没有本质上的区别，只是在它的各成员中，基本上都要用到模板参数类型 T。

例 15.6 定义模式板类。

```
//      EX15_6.H

template<class T>
class   AnyType      {
            T       x, y;
        public:
            AnyType(T a, T b) : x(a), y(b)       {}
            T       GetX() {return x;}
            T       GetY() {return y;}
};
```

头文件 EX15_6.H 定义了一个任意类类型 AnyType，它有两个类型为 T 的私有成员 x，y。当程序利用该模板生成一个对象时，实参的类型就决定了 T 的类型。

例 15.6 所定义的模板类中，所有的成员函数都在模板中被定义成内联的。若在类外定义模板类的成员函数，则必须遵循以下的语法：

```
template<class T>
type   Class_Name<T> :: Function_Name(args)
{
    //...;
}
```

如果模板中带有多个参数，则无论具体的成员函数是否用到它们，所有的参数类型必须在类名后一一列出。

例 15.7 在模板类外定义成员函数。

```
//      EX15_7.H

template<class T, int n = 10>
class   AnyStack     {
        private:
            T       Stack[n];
            int     TopMost;
            int     Top;
        public:
            AnyStack() : TopMost(n), Top(0)    {}
            int     GetTop()       {return Top;}
            int     Push(T);
            int     Pop(T&);
};

template<class T, int n>
int     AnyStack<T, n> :: Push(T elem)
{
```

```
        if(Top > TopMost)
                return 0;
        Stack[Top] = elem;
                Top ++;

        return 1;
}

template<class T, int n>
int     AnyStack<T, n> :: Pop(T &elem)
{
        if(Top == 0)
                return 0;
        Top --;
        elem = Stack[Top];

        return 1;
}
```

头文件 EX15_7.H 中定义了一个抽象的栈类。该类带有一个缺省的整型参数 n，它给出了栈的大小。

15.2.2 使用模板类

使用模板类的第一步就是说明一个模板类的对象。说明模板类的对象时应具有以下的一般形式：

Class_Name<type> Object;

其中：type 是任何一个已存在的数据类型，包括任何实例类。若模板类带有多个参数类型，则除缺省参数外的所有参数都必须给出其实参类型。

例 15.8 使用 AnyType 模板类。

```
//      EX15_8.CPP

#include      <iostream.h>
#include      "ex15_6.h"

void    main()
{
        AnyType<int>        iType(123, 456);
        AnyType<char*>      strType("The C", "The C++");

        cout << iType.GetX() << '\t' << iType.GetY() << endl;
        cout << strType.GetX() << '\t' << strType.GetY() << endl;
}
```

该例的输出为：

```
123   456
The CThe C++
```

例 15.8 中说明了两个 AnyType 类的对象。语句：

```
AnyType<int>          iType(123, 456);
```

说明了一个实参类型为整型的对象 iType 。根据模板类 AnyType 的定义，其成员 x 和 y 将都是 int 型的变量。而语句：

```
AnyType<char*> strType("The C", "The C++");
```

则说明了一个实参类型为字符型指针的对象 strType，该对象的两个数据成员均为字符串。

模板类 AnyStack 是一个非常实用的类，利用该类可以创建用于存放任何类型数据和对象的堆栈。

例 15.9 使用 AnyStack 模板类。

```
//      EX15_9.CPP

#include      <iostream.h>
#include      "ex12_15.h"
#include      "ex15_7.h"

void   main()
{
       AnyStack<int>        iStack;
       AnyStack<float, 5>   fStack;
       AnyStack<String, 3> StrStack;
       char    *str[] = {"C++", "BASIC", "Pascal"};
       int     i, ie;
       float   fe;
       String  Stre;

       for(i = 0; i < 5; i ++)  {
              iStack.Push(i + 1);
              fStack.Push(i * 3.14159);
       }

       for(i = 0; i < 3; i ++)  {
              String  temp(str[i]);

              StrStack.Push(temp);
       }

       while(iStack.Pop(ie))
              cout << ie << "   ";
       cout << endl;

       while(fStack.Pop(fe))
              cout << fe << "   ";
       cout << endl;

       while(StrStack.Pop(Stre))
              cout << Stre << "   ";
```

```
    cout << endl;
}
```

由于栈是一个 LIFO（Last In First Out，后入先出）的数据结构，所以例 15.9 的输出为：

```
5  4  3  2  1
12.56636  9.42477  6.28318  3.14159  0
Pascal  BASIC  C++
```

习题

15-1 设计一个 Swap（）模板函数，该函数可以将任意两个类型相同的变量或对象的值相互交换。

15-2 将习题 14-7 中改进了的数组类改写成模板类。

附录　ASCII 码表

ASCII 值		控制字符	ASCII 值		字符	ASCII 值		字符	ASCII 值		字符
0	00	NUL	32	20	(space)	64	40	@	96	60	`
1	01	SOH	33	21	!	65	41	A	97	61	a
2	02	STX	34	22	"	66	42	B	98	62	b
3	03	ETX	35	23	#	67	43	C	99	63	c
4	04	EOT	36	24	$	68	44	D	100	64	d
5	05	END	37	25	%	69	45	E	101	65	e
6	06	ACK	38	26	&	70	46	F	102	66	f
7	07	BEL	39	27	'	71	47	G	103	67	g
8	08	BS	40	28	(	72	48	H	104	68	h
9	09	HT	41	29	)	73	49	I	105	69	i
10	0A	LF	42	2A	*	74	4A	J	106	6A	j
11	0B	VT	43	2B	+	75	4B	K	107	6B	k
12	0C	FF	44	2C	,	76	4C	L	108	6C	l
13	0D	CR	45	2D	-	77	4D	M	109	6D	m
14	0E	SO	46	2E	.	78	4E	N	110	6E	n
15	0F	SI	47	2F	/	79	4F	O	111	6F	o
16	10	DLE	48	30	0	80	50	P	112	70	p
17	11	DC1	49	31	1	81	51	Q	113	71	q
18	12	DC2	50	32	2	82	52	R	114	72	r
19	13	DC3	51	33	3	83	53	S	115	73	s
20	14	DC4	52	34	4	84	54	T	116	74	t
21	15	NAK	53	35	5	85	55	U	117	75	u
22	16	SYN	54	36	6	86	56	V	118	76	v
23	17	ETB	55	37	7	87	57	W	119	77	w
24	18	CAN	56	38	8	88	58	X	120	78	x
25	19	EM	57	39	9	89	59	Y	121	79	y
26	1A	SUB	58	3A	:	90	5A	Z	122	7A	z
27	1B	ESC	59	3B	;	91	5B	[	123	7B	{
28	1C	FS	60	3C	<	92	5C	\	124	7C	|
29	1D	GS	61	3D	=	93	5D	]	125	7D	}
30	1E	RS	62	3E	>	94	5E	^	126	7E	~
31	1F	US	63	3F	?	95	5F	_	127	7F	

参考文献

[1] 和克智. C++程序设计. 第 1 版. 西安：西安交通大学出版社，1995

[2] 钱子明　应晓. 巧学巧用 Borland C++ 5.0 for Windows 95. 第 1 版. 北京：电子工业出版社，1996

[3] 李振格主编. Borland C++ & Turbo C++程序员手册. 第 1 版，北京：北京航空航天大学出版社，1992

[4] 李振格主编. Borland C++ & Turbo C++库函数手册. 第 1 版，北京：北京航空航天大学出版社，1992

[5] Borland C++ 5.02 Online Help. Borland International, Inc.，1997

[6] Gorlen K.E. . An Object-Oriented Class Library for C++ Programs. Software Practice and Experience 17.　No. 12, 1987

[7] Bruce Eckel. Using C++ Conver C++ Version 2.0. Oskoton McGraw-Hill, 1989

[8] BenEzzell. Turbo C++ Programming. Addison-Wesley Publishing Company, 1990

[9] Tom Swan. Tom Swan's Code Secrets. Sams Publish, 1994

[10] Steve Oualline. Practical C++ Programming. O'Reilly & Associates, Inc., 1995

Document generated by Anna’s Archive around 2023-2024 as part of the DuXiu collection (https://annas-blog.org/duxiu-exclusive.html).

Images have been losslessly embedded. Information about the original file can be found in PDF attachments. Some stats (more in the PDF attachments):

{
  "filename": "MTAyMDM5MDQuemlw",
  "filename_decoded": "10203904.zip",
  "filesize": 20577658,
  "md5": "4ad163826557cae3f0c53c620628e7dd",
  "header_md5": "f5be294c6a21f2dedb7ff81c51cbf59d",
  "sha1": "85b73ce6158bcb56ac058f048779778ecfe42780",
  "sha256": "3d9c05a96cb17512014fae210a54ca8b6e95aaef14c24668b74279bad91d5e5f",
  "crc32": 3811288798,
  "zip_password": "",
  "uncompressed_size": 20974461,
  "pdg_dir_name": "C++\u2502\u2560\u2568\u2265\u2554\u03a6\u255d\u255e C++CHENGXU
SHEJI\u00fa\u00bf\u2561\u250c2\u2591\u00b5\u00fa\u2310_10203904",
  "pdg_main_pages_found": 337,
  "pdg_main_pages_max": 337,
  "total_pages": 349,
  "total_pixels": 2447331144,
  "pdf_generation_missing_pages": false
}